UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

INGENIERÍA INDUSTRIAL:

ELECTRÓNICA Y AUTOMÁTICA

PROYECTO FIN DE CARRERA:

SISTEMA DE EMULACIÓN AUTÓNOMA CON COMUNICACIÓN USB 2.0 INTEGRADO EN UN ENTORNO CON MICROBLAZE

Autor: Pedro Plaza Merino

Tutor: Mario García Valderas

DEPARTAMENTO DE TECNOLOGÍA ELECTRÓNICA

GRUPO DE DISEÑO MICROELECTRÓNICO Y APLICACIONES

SEPTIEMBRE 2010

AGRADECIMIENTOS:

A mis padres quienes, hicieron de mi lo que soy,
y a los que les debo todo.

A mi hermana, que me ha apoyado
incondicionalmente siempre.

A Mariluz, mi amiga y novia,
una persona muy especial, quién me ha ayudado
mucho en la última etapa de la carrera.

A Mario, mi tutor, sin él
este proyecto no habría sido posible.

A mis amigos y compañeros, por estar siempre
que les he necesitado, a mi lado.

A todos y cada uno de los integrantes del
Grupo de Diseño Microelectrónico y Aplicaciones.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ÍNDICE DE ACRÓNIMOS

ÍNDICE DE ACRÓNIMOS:

ASIC: Application-Specific Integrated Circuit.

BER: Bit Error Rate.

BSB: Base System Builder.

CMDA: Code division multiple access.

CRSC: Circular Recursive Systematic Convolutional.

CUT: Circuit Under Test

DMA: Direct Memory Access.

DVB-RCS: Digital Video Broadcasting - Return Channel via Satellite.

EMC: External Memory Controller.

FIFO: First In First Out.

FF: Flip Flop.

FPGA: Field Programmable Gate Arrays.

GUI: Graphic User Interface.

HDL: Hardware Description Language.

IC: Integrated Circuit.

MB: MicroBlaze.

OPB: On-chip Peripheral Bus.

PC: Personal Computer.

PHY: Physical Layer.

RAM: Random Addressable Memory.

RISC: Reduced Instruction Set Computer.

SEU: Single Event Upset.

UART: Universal Asynchronous Receiver-Transmitter.

UMTS: Universal Mobile Telecommunications System.

USB: Universal Serial Bus.

VHDL: VHSIC Hardware Description Language.

VHSIC: Very High Speed Integrated Circuit.

VLSI: Very Large Scale Integration.

INTRODUCCIÓN

1. INTRODUCCIÓN

1.1. ÁMBITO DEL PROYECTO

Este proyecto ha sido realizado en el Departamento de Tecnología Electrónica, dentro del Grupo de Diseño Microelectrónico y Aplicaciones.

La Real Academia Española en su diccionario define Microelectrónica como "Técnica de diseñar y producir circuitos electrónicos en miniatura, aplicando especialmente elementos semiconductores.". La Microelectrónica, también se puede definir como el conjunto de ciencias y técnicas con las que se realizan y fabrican circuitos electrónicos, sobre una pastilla de un semiconductor, lo que formará un circuito integrado (IC)

El enfoque de este proyecto está centrado en la tolerancia a fallos frente a radiación. El motivo del estudio de la tolerancia a fallos frente a radiación, ya que existe un efecto de la radiación sobre los circuitos electrónicos denominado "SEU", del inglés Single Event Upset, y produce un mal funcionamiento de los mismos. El efecto del SEU hace que, a un circuito en funcionamiento normal, cuando impacta radiación sobre sus componentes de memoria, éstos en lugar de tener almacenado en la posición de memoria un "cero lógico", pasan a presentar un "uno lógico", con lo que el contenido del elemento de memoria se ve alterado.

Para estudiar lo tolerantes que son los circuitos diseñados frente a la radiación se plantea la inyección de fallos a circuitos electrónicos. Hay tres mecanismos clásicos para realizar la inyección de fallos: inyección física, mediante simulación y, por último, la emulación de fallos en circuitos electrónicos. Este Proyecto Fin de Carrera se enmarca en el último método para evaluar la tolerancia frente a fallos, mediante un Sistema de Emulación Autónoma.

Estos circuitos electrónicos a los que se les inyectan fallos, en su vida útil, son susceptibles de sufrir fallos debido a causas tales como radiación, cuando son empleados en ambientes tales como el espacio exterior. Una de las preocupaciones de los diseñadores de estos circuitos es la tolerancia frente a fallos de los circuitos que diseñan, lo que mitigan con diversas técnicas de diseño tales como el endurecimiento de hardware. Pero, con estos métodos no se asegura al cien por cien el buen funcionamiento de los circuitos diseñados que vayan a ser utilizados en aplicaciones como son las aeroespaciales. Por eso, un objetivo muy importante en la fase de diseño es conocer cuantitativamente la vulnerabilidad que presentan frente a fallos.

La evaluación de la tolerancia a fallos se realiza con dos copias del circuito a analizar. Por un lado un circuito al que no se le inyectará fallos, denominado Golden, y otra copia exactamente igual, a la que se inyectarán fallos, denominado Faulty. A ambos circuitos se les excitará con las mismas entradas y se realizará una comparativa de las salidas que generan cada uno de ellos. De esta manera se podrá clasificar el tipo de fallo que se le ha inyectado observando cómo afecta a sus salidas y el nivel de propagación que puede tener un fallo inyectado en un elemento de memoria del circuito. La inyección de fallos en un circuito se denomina campaña y presenta dos variables principales a la hora de realizarla: el número de ciclos del banco de pruebas empleados y en nuestro caso el número de biestables (como elemento de memoria) en los que se inyectan fallos. El número de ciclos del banco de pruebas es el número de excitaciones, en forma de entradas que se le aplican al circuito. La campaña transcurre, primero inyectando un fallo al primer biestable, y manteniendo ese fallo se le van excitando las entradas y analizando las salidas, comparando las salidas de la copia buena, Golden con las salidas de la copia mala, Faulty. Pasando posteriormente a la inyección de un fallo al siguiente biestable y analizando de nuevo el comportamiento de ambas copias completando el ciclo para esta nueva inyección. Así

sucesivamente hasta que se hayan realizado todos los ciclos de la campaña de inyección de fallos. Mediante un diccionario de fallos se clasificarán los fallos inyectados en función de la repercusión que hayan tenido en el circuito. En el ámbito de este proyecto se tienen en cuenta tres tipos de fallos: Fallos, Latentes y Silencios. Los fallos clasificados como Fallos desembocan en un mal funcionamiento del circuito. Los fallos Latentes son aquellos que no afectan al funcionamiento del circuito, pero se propagan a lo largo de los elementos de memoria del circuito. Y por último, los fallos Silenciosos son los que ni son detectados en el funcionamiento del circuito, ni permanecen por los elementos de memoria.

En el ámbito de la inyección de fallos, la emulación con FPGAs es la más rápida junto a la inyección de fallos física. Pero el sistema de Emulación Autónoma es el método más rápido y que presenta un coste menor en la realización de campañas de fallos.

Por ejemplo: un circuito con 1000 biestables, que se somete a 5000 ciclos de banco de pruebas, puede suponer 5 millones de fallos inyectados, lo que podría presentar, teniendo en cuenta un circuito con un reloj de entrada de 50 MHz, y una velocidad de transmisión de datos de 115200 baudios, y teniendo que transmitir 10 registros de 32 bits cada uno al ordenador, se obtiene un tiempo de emulación de 3 horas y 52 minutos aproximadamente. La misma campaña con un sistema de simulación de fallos podría requerir al menos una semana.

Dentro de los diferentes sistemas de inyección de fallos a circuitos nombrados anteriormente, el que presenta una mejor relación velocidad/coste, es la inyección de fallos basada en la emulación de los circuitos mediante la descripción de hardware de los mismos, utilizando como soporte físico FPGAs.

El Sistema de Emulación Autónoma del que se parte al inicio de este proyecto está formado por una arquitectura que contiene cuatro grandes bloques: circuito bajo test, control y UART, como se muestra en la Figura 1.1.

El CUT, Circuit Under Test, está formado por una copia del circuito a probar, a la que no se le inyectan fallos, denominada Golden y otra copia a la que sí se le inyectan fallos, denominada Faulty. Introduciendo el mismo juego de entradas a ambas copias, se analizan las salidas de ambos circuitos, de manera que se identificará la inmunidad del circuito frente a los fallos que se le inyectan a sus componentes de memoria.

La UART es elemento que se emplea para las comunicaciones del Sistema de Emulación Autónoma con el exterior, utilizando el protocolo RS-232, conectando la tarjeta al puerto serie de un ordenador. Dentro de la unidad de recepción y transmisión asíncrona, se encuentran unos registros de configuración que contienen la información referente a la campaña que se realizará si el sistema comienza a realizar una emulación.

Por último el bloque de control, es el encargado de gestionar los datos que se envían y reciben a través de la UART, y de realizar las campañas de inyecciones de fallos, clasificar los fallos detectados con un diccionario de fallos y almacenar una copia temporal de los resultados que serán enviados a un ordenador conectado a la tarjeta en la que se implemente el Sistema de Emulación Autónoma.

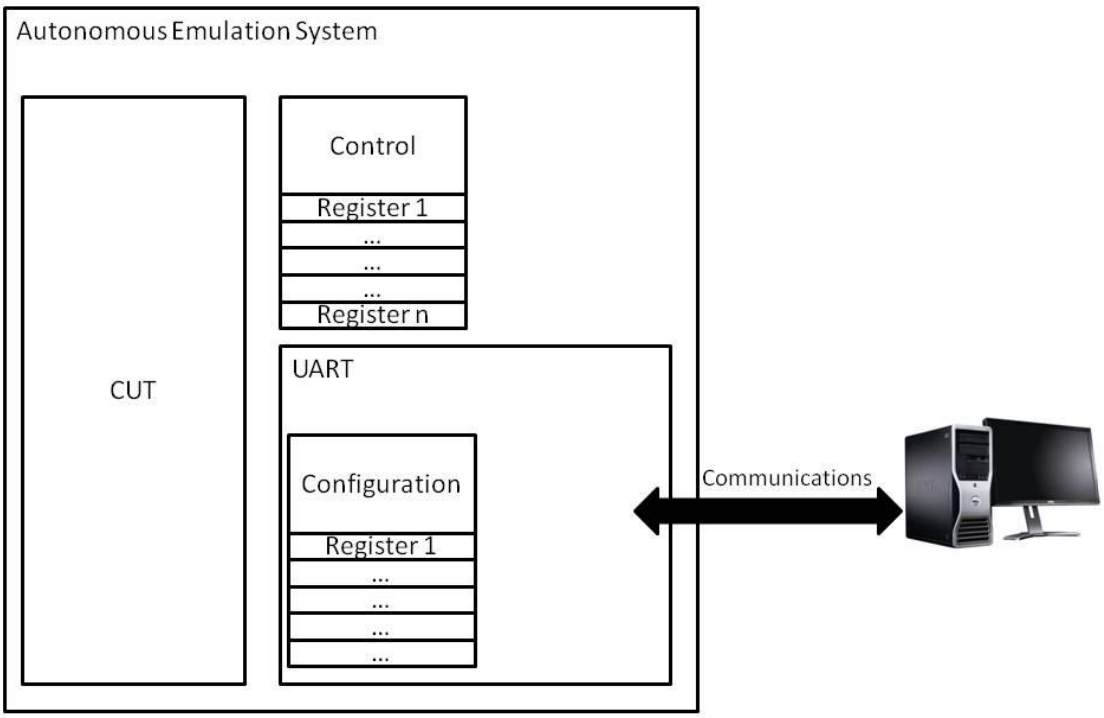

Figura 1.1: Arquitectura del Sistema de Emulación Autónoma.

1.2. OBJETIVOS

El punto de partida es un sistema de Emulación Autónoma, que emplea una UART con comunicación serie, con el protocolo RS-232. Como se aprecia en el ejemplo anterior este tipo de comunicaciones presenta un inconveniente, las comunicaciones son un cuello de botella en la emulación de circuitos electrónicos, ya que los resultados son generados a más velocidad de lo que son transmitidos y posteriormente almacenados. Por ello se plantea como objetivo principal realizar un sistema de Emulación Autónoma, que sea capaz de enviar los resultados a mayor velocidad de lo que son generados los resultados de emulación.

Para poder acometer este objetivo se ha optado por una solución, implementar las comunicaciones USB incluyendo al sistema un entorno con un microcontrolador empotrado de la empresa Xilinx, el MicroBlaze. Un inconveniente a esta solución es el impedimento de no poder usar un programa tipo HyperTerminal, como el usado para las comunicaciones empleadas en el sistema de Emulación Autónoma del que se parte, el Tera Term Web 3.1. Para poder cumplir con el objetivo principal se plantea desarrollar una aplicación de interfaz de usuario para gestionar las comunicaciones USB y almacenar los resultados de la emulación recibidos, además de tratar los resultados generando un archivo donde se muestren los mismos.

De manera que los objetivos de este Proyecto Fin de Carrera se resumen a continuación:

- Construir un emulador.
- Integrar ese emulador en un entorno en el que haya un microcontrolador empotrado MicroBlaze.
- Añadir la funcionalidad de comunicaciones USB.

- Desarrollar una interfaz gráfica que se encargue de:
    - Las comunicaciones USB entre ordenador y tarjeta.
    - Almacenar los resultados recibidos por USB.
    - Mostrar estadísticas en tiempo real de la emulación en curso.
    - Generar archivos de resultados, uno para ser analizado por el usuario y otro para poder tratar los resultados en hojas Excel.

1.3.BREVE DESCRIPCIÓN DE LA MEMORIA

La memoria está dividida en tres grandes bloques: el sistema de Emulación Autónoma, el microcontrolador empotrado MicroBlaze y la Interfaz Gráfica de Usuario.

A lo largo del segundo capítulo se describen, tanto el Hardware empleado en este proyecto, como las aplicaciones que han sido empleadas a lo largo del desarrollo del proyecto.

En el siguiente capítulo se realiza una introducción al sistema de Emulación Autónoma que se desea implementar, dando una pequeña introducción al sistema de Emulación Autónoma del que se parte y del circuito bajo pruebas que se ha empleado.

En el cuarto capítulo se explica el entorno relacionado con el microcontrolador MicroBlaze (MB), comentando las funciones implementadas en el código que ejecutará. Y el sistema formado por el MB y todos los dispositivos que este manejará para realizar las comunicaciones USB con el ordenador, recibir los parámetros de emulación y transmitir los resultados de la emulación, y las comunicaciones con el emulador, es decir, pasarle los parámetros de emulación y recibir los resultados de emulación.

El siguiente capítulo, describe todas las funciones que se han desarrollado para realizar la aplicación que hará de intérprete entre el usuario y el sistema de Emulación Autónoma utilizando comunicación USB entre el PC y la tarjeta empleada. Además de todas las funcionalidades desarrolladas en torno al tratamiento de los resultados y la configuración de la aplicación.

El sexto capítulo define el sistema completo, desde las conexiones entre el ordenador y la tarjeta, pasando por la carga del archivo .bit, bitstream, en la FPGA y concluyendo con una breve guía con los pasos necesarios para poder realizar una emulación de un circuito electrónico.

ENTORNO DE TRABAJO

2. ENTORNO DE TRABAJO

A lo largo de este capítulo se explican brevemente las herramientas empleadas tanto de Hardware como de Software empleadas para la realización del proyecto. En primer lugar el Hardware de desarrollo utilizado, una tarjeta de propósito general con una FPGA y diversos periféricos puesta en el mercado por la empresa AVNET. Para el desarrollo Hardware se ha empleado una herramienta de síntesis, el Symplify y el Xilinx Platform Studio con las que se desarrolla todo el entorno relativo a la FPGA. Y por último el Visual Studio con el que se ha desarrollado la interfaz gráfica de usuario para las comunicaciones USB.

2.1.HARDWARE DE DESARROLLO

Para el desarrollo de este Proyecto Fin de Carrera se ha utilizado la tarjeta "Xilinx Virtex 4 LX Evaluation Kit". El Grupo de Diseño Microelectrónico y Aplicaciones cuenta con varias tarjetas de desarrollo de propósito general, ésta es la que más se aproxima a las necesidades de este proyecto. El núcleo de esta tarjeta es una FPGA Virtex 4 con 59.904 celdas lógicas, además tiene diversos periféricos para poder afrontar diversos proyectos. Los periféricos que posee y que son utilidad para el proyecto que se está tratando son el puerto de comunicaciones RS-232 y el controlador USB 2.0.

A continuación se exponen sus componentes:

- FPGA:
    - FPGA Virtex-4 XC4VLX60-FF668.
- Periféricos de Entrada/Salida:
    - Display gráfico de 128x64 OSRAM.
    - Conectivididad AvBus incluyendo parejas 30 LVDS.
    - 8 Interruptores de posición tipo DIP.
    - 2 botones tipo pulsador.
    - Parrilla de 8 LEDs.
- Memoria:
    - Memoria DDR SDRAM de Micron de 32 MB.
    - Memoria FLASH de 8 MB.
    - Memoria FLASH de Xilinx.
- Comunicaciones:
    - Puerto serie RS-232.
    - Ethernet PHY DP83847 10/100 de National Semiconductor.

  - Controlador USB 2.0 de Cypress, CY7C68013.

- Control de frecuencia:
  - Multiplicador/divisor de reloj CDC5801 de Texas Instruments.
- Alimentación:
  - Módulo de alimentación de Texas Instruments.
  - Reguladores lineales de National Semiconductor.
- Configuración:
  - JTAG para programar la FPGA con un cable USB.

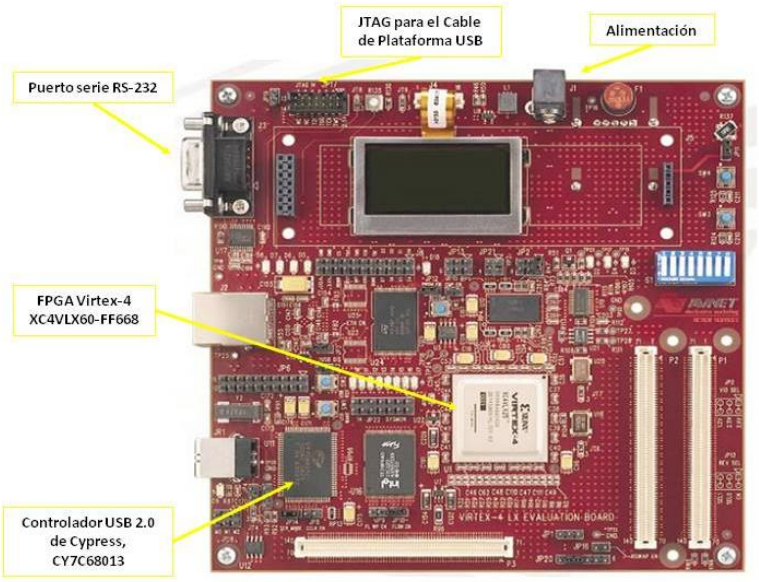

Figura 2.1: Xilinx Virtex 4 LX Evaluation Kit.

La FPGA de que dispone la tarjeta de evaluación es una Virtex 4 LX60 con las siguientes características:

Device	Array Row x Col	Logic Cells	Slices	Distributed RAM (Kb)	XtremeDSP Slices	18 Kb Blocks	Block RAM (Kb)	DCMs	PMCDs	I/O Banks	User I/O
XC4VLX60	128 x 52	59,904	26,624	416	64	160	2,880	8	4	13	640

Tabla 2.1: Características de la FPGA Virtex LX60

El periférico USB de alta velocidad CY7C68013 de Cypress tiene las siguientes características a destacar:

- Transceptor USB 2.0, motor de interfaz serie y un microprocesador 8051 mejorado, todo integrado en un solo chip.
- Software: es ejecutado desde un memoria RAM interna, que se puede:
    - Descargar por USB.
    - Cargar desde una memoria EEPROM.
- Cuatro canales de comunicación programables BULK/INTERRUPT/ISOCHRONOUS.
- Intercambio de datos externos de 8- o 16-bit.

2.2. HERRAMIENTAS HARDWARE

Para construir un sistema de Emulación Autónoma de fallos es necesario en primer lugar realizar una síntesis, obteniendo una netlist descrita en leguaje VHDL del mismo. Posteriormente se someterá a un proceso de instrumentación del circuito con el que se obtendrá la copia Golden y la Faulty empleadas para la emulación. Para poder instrumentar el circuito es necesario que la netlist no presente diseño jerárquico por lo que se ha optado por una herramienta que lo permita, la aplicación Synplify de la empresa Synplicity, mostrada en la Figura 2.2

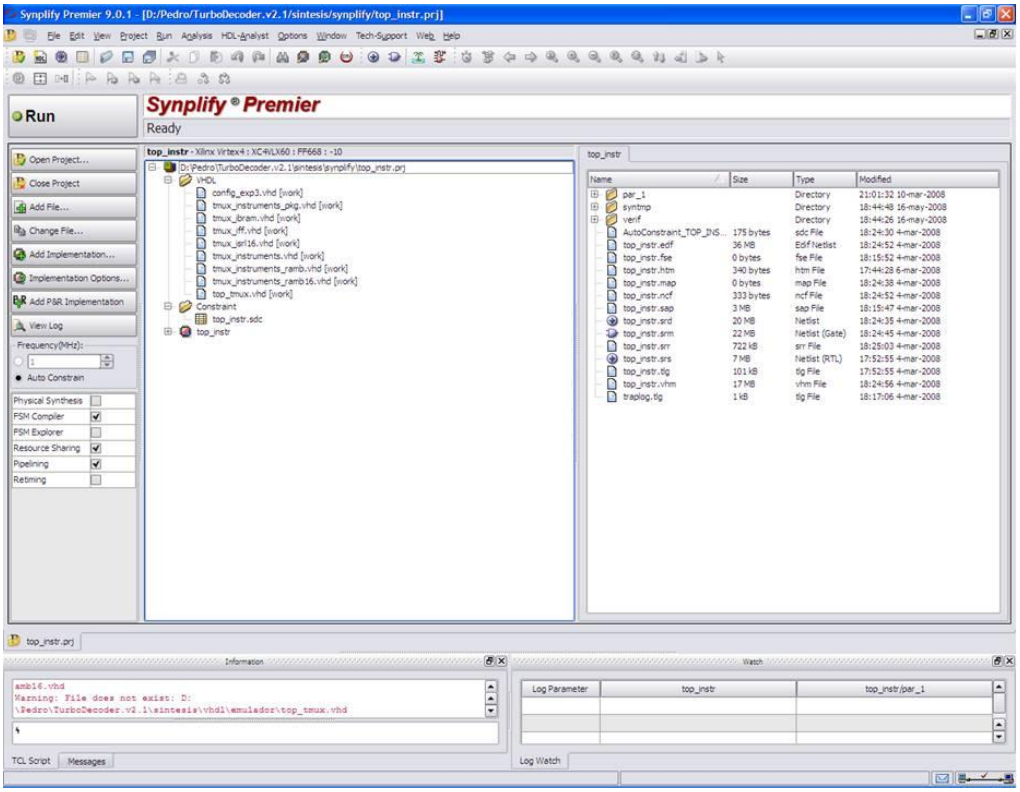

Figura 2.2: Entorno de desarrollo Synplify de Synplicity.

Todo el desarrollo implementado en la tarjeta "Xilinx Virtex 4 LX Evaluation Kit" ha sido desarrollado con la herramienta "Xilinx Platform Studio".

Esta herramienta desarrollada por Xilinx permite diseñar circuitos en los que se pueden integrar microprocesadores embebidos como el PowerPC ó el MicroBlaze. Permite desarrollar diseños de circuitos con microcontroladores embebidos en FPGA muy rápidamente, de manera que el tiempo empleado en integrar todos los componentes se ve reducido sustancialmente.

Con esta herramienta se ha integrado el microcontrolador embebido MicroBlaze, el driver para realizar las comunicaciones USB y una UART de comunicaciones series empleada para depuración en la fase de desarrollo. Como se muestra en la Figura 2.3, esta aplicación presenta una interfaz muy intuitiva lo que acelera el proceso de desarrollo referente a las conexiones entre el microcontrolador y los periféricos empleados.

Para la integración del emulador de fallos se ha construido un periférico que consta de 24 registros mediante los que se comunican el emulador de fallos y el MicroBlaze.

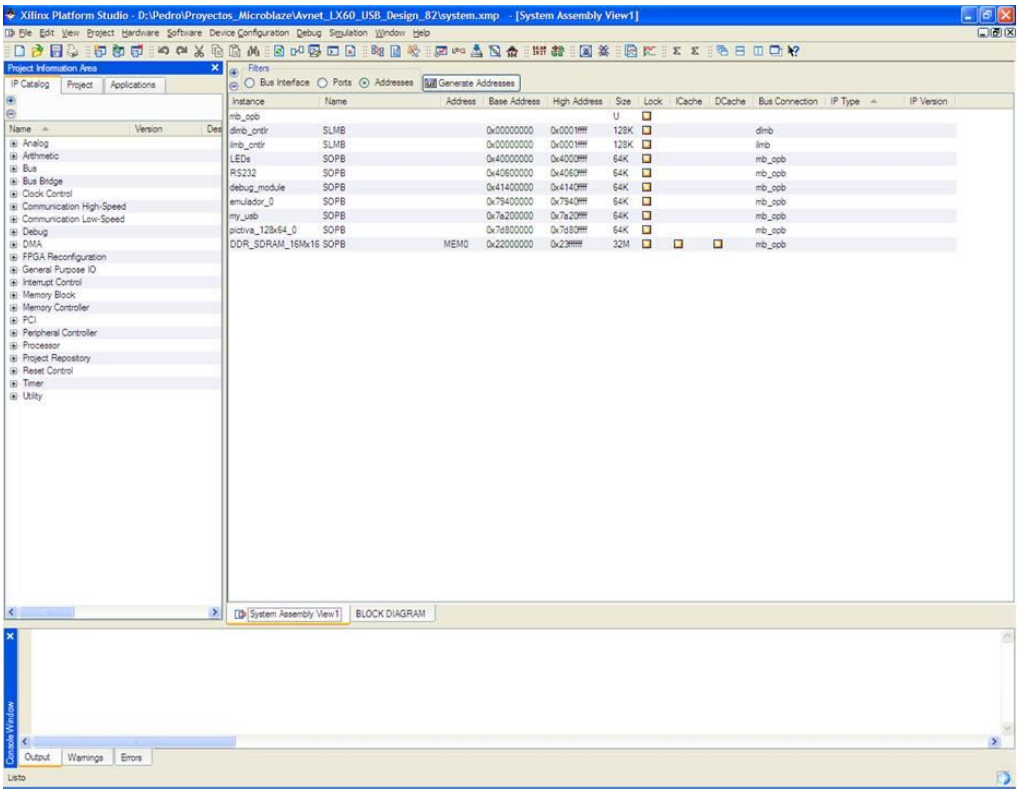

Figura 2.3: Entorno de desarrollo Xilinx Platform Studio.

2.3. HERRAMIENTAS SOFTWARE

Para la implementación de la interfaz gráfica de usuario que gestiona las comunicaciones USB entre el PC y el MicroBlaze se ha utilizado el "Visual Studio 2008", mostrado en la Figura 2.4, programa mediante el que se pueden desarrollar aplicaciones de interfaz gráfica bajo sistemas operativos Windows. Ha sido necesario su uso ya que no existe ninguna aplicación comercial para poder comunicar mediante USB la tarjeta y el PC.

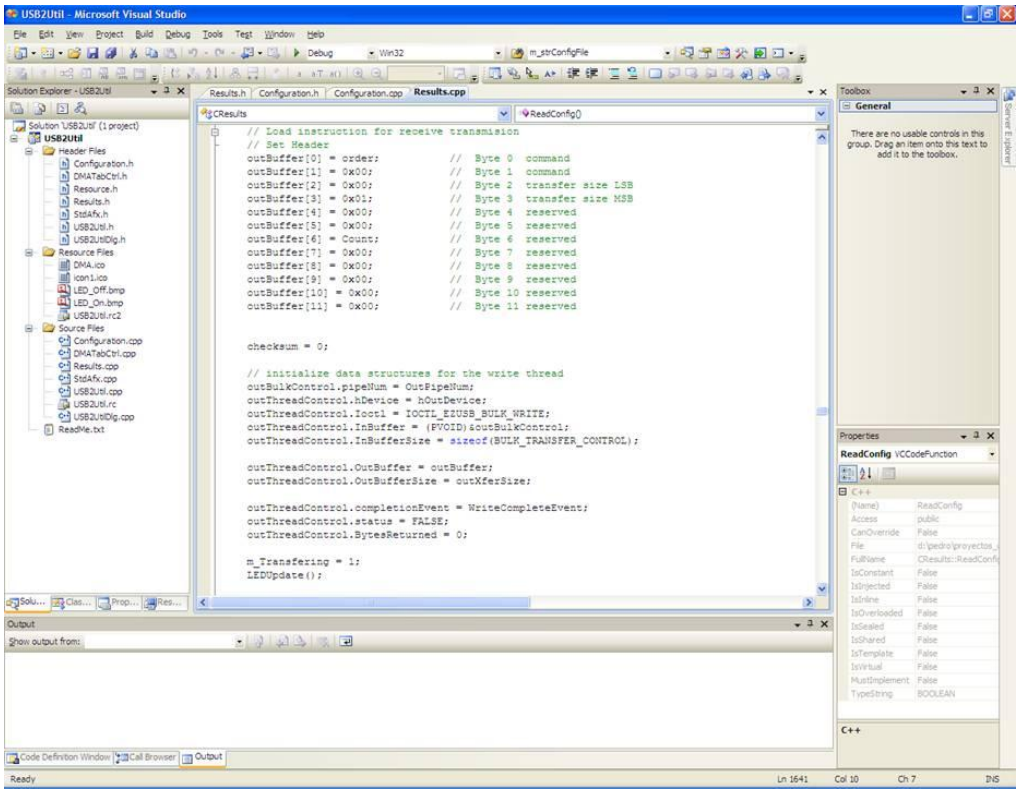

Figura 2.4: Entorno de desarrollo Visual Studio 2008.

DESCRIPCIÓN DEL SISTEMA

3. DESCRIPCIÓN DEL SISTEMA

A lo largo de este capítulo se expone la descripción del sistema de partida y los medios empleados para poder cumplir con los objetivos del proyecto. El punto de partida es un sistema de emulación autónoma con un canal de comunicaciones lento, lo que implicaría que en campañas realizadas se puedan perder datos obteniendo unos resultados de análisis incompletos y poco realistas. Para ello se ha optado por implementar un canal de comunicaciones USB, basándose en un entorno con un microcontrolador embebido que facilita Xilinx y los componentes necesarios. Lo que facilita la etapa de desarrollo del proyecto, construyéndolo en un entorno modular.

3.1. SISTEMA DE EMULACIÓN AUTÓNOMA

3.1.1. INTRODUCCIÓN

Los circuitos electrónicos que trabajan en entornos con radiación son propensos a sufrir cambios de estado en sus componentes de memoria, denominados, SEU. Por este motivo, los circuitos de integración a gran escala, circuitos VLSI, del inglés Very Large Scale Integration, es necesario endurecerlos contra este tipo de fenómeno, cuando se desarrollan para su funcionamiento en entornos donde pueden recibir radiación y de esta forma alterarse su funcionamiento normal.

Este efecto es un cambio del estado del componente causado por el impacto de iones o radiación electromagnética en un nodo sensible de un dispositivo electrónico, como puede ser un microcontrolador, una FPGA, transistores de potencia. Los efectos de los SEU aparecen en los elementos de memoria de los circuitos.

Una tarea importante en el diseño de este tipo de circuitos es la estimación de la sensibilidad de los mismos frente a los efectos de los SEUs, lo que permite predecir la tasa de error del circuito. Esta tasa de error es necesaria para identificar áreas débiles que deben ser endurecidas para validar la efectividad del diseño.

Para este propósito, es necesario poder aplicar las técnicas y herramientas, que permitan estimaciones fiables de la sensibilidad frente a SEU de una forma rápida y no muy costosa.

La inyección de fallos ha sido ampliamente aceptada para llevar a cabo los análisis de sensibilidad frente a SEU. Dentro de la inyección de fallos se encuentran varios tipos:

- Exponer el circuito a radiación, lo que produce resultados muy realistas.
- Inyección de fallos con láser.
- Inyección de fallos con interferencias electromagnéticas.

El inconveniente que presentan los tres tipos anteriores es que requieren un equipo muy caro y fabricar un circuito específico para llevar a cabo el test.

La inyección de fallos basada en la simulación es capaz de facilitar muchos modelos de fallos y aporta una gran flexibilidad a la hora de configurar campañas de inyección de fallos. La inyección de fallos se realiza a través de simuladores HDL, Hardware Description Language. Mediante comandos del simulador, ó modificando el modelo HDL para incluir la capacidad de

inyección de fallos. El inconveniente de esta técnica es un coste computacional prohibitivo que presenta ante un gran número de inyecciones de fallos.

En los últimos años ha surgido una nueva técnica, la inyección de fallos basada en la emulación. Ya que el diseño y realización de circuitos ASIC, del inglés Application-Specific Integrated Circuit, es un proceso lento y que implica un elevado coste, lo que es un problema, nace la alternativa de la emulación de circuitos electrónicos como solución a este inconveniente. Los circuitos integrados de aplicaciones específicas pueden ser descritos mediante un leguaje de descripción de Hardware, como el VHDL y utilizar dispositivos FPGA, Field Programmable Gate Arrays, para implementarlos. Obteniendo un sistema de emulación dónde se le inyectan fallos, para analizar la tolerancia a fallos del circuito.

El propósito de la inyección de fallos basada en la emulación es evaluar la sensibilidad frente a SEU de los diseños de circuitos ASIC, no de un diseño FPGA.

Una gran ventaja de las FPGA es una relación velocidad/coste de las emulaciones de inyección de fallos frente a otras técnicas. Lo que permite inyectar muchos fallos para analizar la respuesta del circuito en muy poco tiempo, comparado con los sistemas de emulación basados en simulaciones. Y frente al resto de sistemas de inyección de fallos, es que no requiere fuertes inversiones en equipos, ni que sea necesario fabricar un circuito específico para llevar a cabo el test.

Un factor muy importante en el sistema de emulación autónoma es, que necesita un canal de comunicaciones entre la FPGA y el host, PC, ya que el ordenador es el encargado de ir almacenando los resultados de la emulación de forma continua. Esto podría ser un inconveniente, ya que el canal de comunicación, en algunas emulaciones se puede convertir en un cuello de botella, con la consecuente pérdida de datos.

Retomando el ejemplo del capítulo 1, un circuito con 1000 biestables, con un banco de pruebas que consta de 5.000 ciclos de reloj, puede suponer 5 millones de fallos inyectados, lo que podría presentar, teniendo en cuenta un circuito con un reloj de entrada de 50 MHz, y una velocidad de transmisión de datos de 115200 baudios, y teniendo que transmitir 10 registros de 32 bits cada uno al ordenador, se obtiene un tiempo de emulación de 3 horas y 52 minutos aproximadamente. Tiempo necesario para la obtención de los resultados, pero los resultados de toda la campaña tardan el generarse aproximadamente 3 minutos y 20 segundos. De lo que se obtendría una clara pérdida de datos.

El sistema de emulación de fallos es el encargado de la inyección de los fallos en el circuito a probar y de la recopilación de los resultados obtenidos en la campaña, enviándolos al exterior para que puedan ser analizados. Se implementa en una FPGA mediante la descripción de Hardware de todo el sistema y es controlado a través de un ordenador. Consta de tres grandes bloques: el CUT, del inglés Circuit Under Test, el bloque de control y la UART unidad asíncrona re recepción y transmisión que emplea el protocolo de comunicaciones serie RS-232.

El CUT incluye el circuito que se desea probar y está formado por una copia del circuito a la que no se le inyectarán fallos, la que se utilizará como referencia de un correcto funcionamiento del circuito, denominada Golden. La copia a la que se le inyectan los fallos es denominada Faulty.

El módulo de control es el encargado de gestionar los parámetros de la campaña de inyección de fallos que se va a llevar a cabo en un banco de registros que se escriben antes de comenzar el proceso de emulación, también presenta unos registros donde se almacenan los resultados de la emulación que serán enviados al exterior. También de traducir los resultados de los fallos inyectados mediante un diccionario de fallos, de almacenar los resultados obtenidos y de dar la

orden de inicio de la emulación, con la consecuente gestión del proceso de emulación autónoma, lo que conlleva realizar la inyección del fallo en el correspondiente elemento de memoria, y compara las salidas obtenidas de las dos copias del circuito, de la Golden y de la Faulty.

El componente UART es el encargado de gobernar las comunicaciones del sistema con el exterior, un equipo informático conectado al emulador de fallos mediante un puerto serie. Este componente incluye una batería de registros donde se almacenan tanto los datos que salen, cómo los datos que entran al emulador.

A continuación en la siguiente figura, Figura 3.1, se muestra un la arquitectura del sistema de emulación autónoma donde se representan sus partes: el CUT, el módulo de control, el módulo de comunicaciones con el exterior, el PC además se incluye la ubicación de los registros de configuración, de estado y de resultados situados en el módulo de control, y los registro que necesita el módulo de comunicaciones.

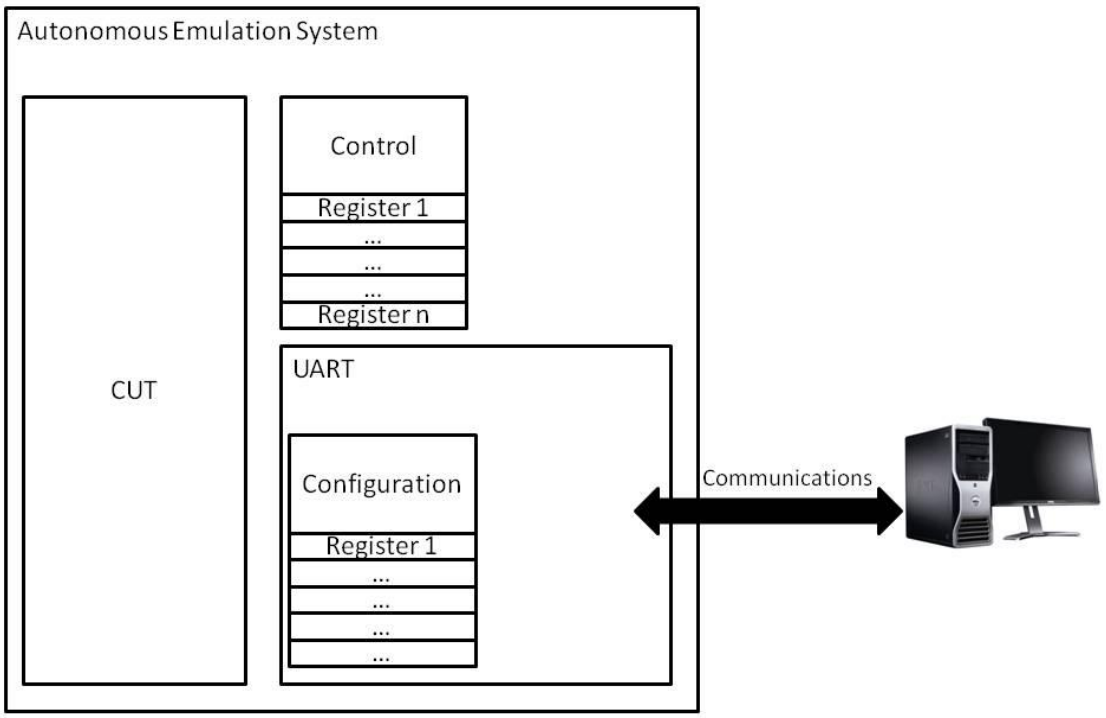

Figura 3.1: Arquitectura de un sistema de emulación autónoma.

3.1.2. CIRCUITO BAJO TEST

El circuito que se prueba con el sistema de emulación autónoma es un turbodecodificador, "TurboDecoder". Circuitos como este son empleados en satélites que se encuentran en misiones de exploración espacial, ó para intercambiar información entre satélites y la Tierra. Estos circuitos decofican turbo códigos, denominados "Turbo Codes".

Los turbo códigos, "Turbo Codes", originalmente en francés, "Turbocodes" tienen como característica principal su impresionante capacidad de corrección de errores, lo que posibilita comunicaciones con una tasa de error, BER, Bit Error Rate, cercanas a la capacidad del canal. En los últimos años una gran atención en los campos de la teoría de la información y las telecomunicaciones, siendo su inclusión en estándares de última generación una consecuencia directa de su excelente desempeño. En particular, esta clase de códigos de corrección de errores

ha sido la elegida en la implementación de los dos más importantes estándares de comunicación de telefonía móvil de tercera generación (UMTS y CDMA2000).

El turbo decodificador estudiado ha sido diseñado por la empresa Thales Alenia Space, Cumple estándar DVB-RCS, Digital Video Broadcast-Return Channel by Satellite, es un estándar abierto definido en mayo de 2000, para permitir las aplicaciones interactivas y bidireccionales a través de las redes y sistemas satelitales. De esta manera la transmisión y recepción de los datos se realiza a través de enlaces vía satélite, y sin necesidad de un canal de retorno terrestre. El estándar DVB-RCS ofrece una solución normalizada para integrar el canal de retorno en la red satélite. Generalmente se basa en sistemas con topología de estrella, donde existe un hub central, el satélite, y múltiples terminales receptores y trasmisores en la superficie terrestre. El circuito a analizar El módulo turbo decodificador ha sido diseñado centrándose en cinco de las siete posibles tasas que se definen en el estándar DVB-RCS.

Como ya se ha mencionado antes, los efectos de la radiación pueden añadir niveles de ruido en los módulos de comunicación sin cables, incluso utilizando turbo códigos, los que aseguran unas bajas tasas de error (BERs). Por eso, la robustez de los circuitos diseñados debe analizarse, y en particular el efecto de los SEUs en la tasa de error, por eso se emplea una herramienta de evaluación autónoma, haciendo posible una intensiva inyección de fallos en el módulo turbo decodificador.

La sensibilidad frente a SEUs del turbo decodificador se evalúa configurando cuatro parámetros distintos que son los siguientes:

- Rate: la tasa de código es la relación entre la longitud del mensaje original (k bits) y la longitud del código completo, que consta de un número de dado de bits (n bits), donde la diferencia entre el tamaña del código completo y el tamaño del mensaje original son los bits de paridad (n-k bits).
- Length: Éste parámetro fija el tamaño de los paquetes que se mandan. Por ejemplo el paquete contiene 256 bytes y 1048 símbolos.
- Offset: Está relacionado con la tasa de generación de ruido.
- Iterations: El número de iteraciones que necesita el turbo decodificador para decodificar un paquete.

3.2. APLICACIÓN ESPECÍFICA CON MICROBLAZE

Dado que el sistema de emulación autónomo del que se parte requiere una tasa de transferencia mayor para no perder resultados se ha optado por implementar un módulo de comunicaciones USB al sistema. Para poder acometer este objetivo se ha elegido utilizar como base la posibilidad que ofrece Xilinx con sus microprocesadores embebidos, facilitando la etapa de desarrollo, permitiendo la utilización de núcleos de propiedad intelectual, IP Cores, que son descripciones Hardware de circuitos. De manera que se puede construir un sistema de manera modular.

Se ha elegido el MicroBlaze ya que presenta una relación de capacidad de cálculo frente a ocupación en la FPGA muy bueno y adecuado a las necesidades de este proyecto. Siendo fácil y rápido el desarrollo de un proyecto base donde se incluyan los componentes necesarios para poder implementar las comunicaciones USB en el sistema de emulación de fallos.

Para el control de las comunicaciones entre el sistema de emulación autónoma y el PC se ha utilizado un microcontrolador embebido en la FPGA, un MicroBlaze.

Además Xilinx facilita un módulo que se encarga de gestionar las comunicaciones entre los distintos elementos que componen el sistema integrado alrededor del MicroBlaze.

El modulo de arbitraje OPB se usa para interconexiones en sistemas basados en microcontroladores embebidos de FPGAs de Xilinx. Es una implementación de distribución multiplexada como una función AND en el maestro o esclavo accediendo al bus, y una función OR para combinar los componentes en un solo bus. Presenta las siguientes características:

- Incluye arbitraje parametrizado OPB.
- Incluye señales de Entrada/Salida parametrizadas para soportar hasta 16 maestros y cualquier número de esclavos. Xilinx recomienda hasta 16 esclavos como máximo.
- Incluye todas las señales presentes en la especificación OPB V2.0 excepto para señales handshake de DMA.
- La estructura OR puede ser implementada usando solamente LUTs o puede usarse una combinación de LUTs y sumadores rápidos con acarreo para reducir el número de LUTs empleadas en las interconexiones OR.
- Incluye 16 relojes con reset al encendido, y parámetros para reset externo del bus a nivel alto ó bajo.
- Incluye una entrada para reset desde el Watchdog Timer.

A este bus se conectan todos los periféricos del sistema para comunicarse con el MicroBlaze. El microcontrolador toma el papel de maestro en el bus intercambiando información con los periféricos que utilizan el bus en modo esclavo.

A nivel de software tanto los periféricos como el MicroBlaze escriben o leen unos registros, los que tienen unas direcciones fijadas, se utilizan buffers para almacenar la información que quieren intercambiarse, las direcciones de esos buffers se muestran en la tabla 3.1:

Nombre del Periférico	**Dirección Inicial**	**Dirección Final**	**Tamaño**
LEDs	0x40000000	0x4000FFFF	64KB
RS232	0x40600000	0x4060FFFF	64KB
Debug_module	0x41400000	0x4140FFFF	64KB
emulador_0	0x79400000	0x7940FFFF	64KB
My_usb	0x7A200000	0x7A20FFFF	64KB
Pictiva_128x64_0	0x7D800000	0x7D80FFFF	64KB
DDR_SDRAM_16Mx16	0x22000000	0x23FFFFFF	32M

Tabla 3.1: Direcciones de los periféricos en el bus OPB.

La arquitectura del sistema de emulación al que se quiere llegar, implementado con el microprocesador MicroBlaze dentro de la FPGA, se muestra en la Figura 3.2.

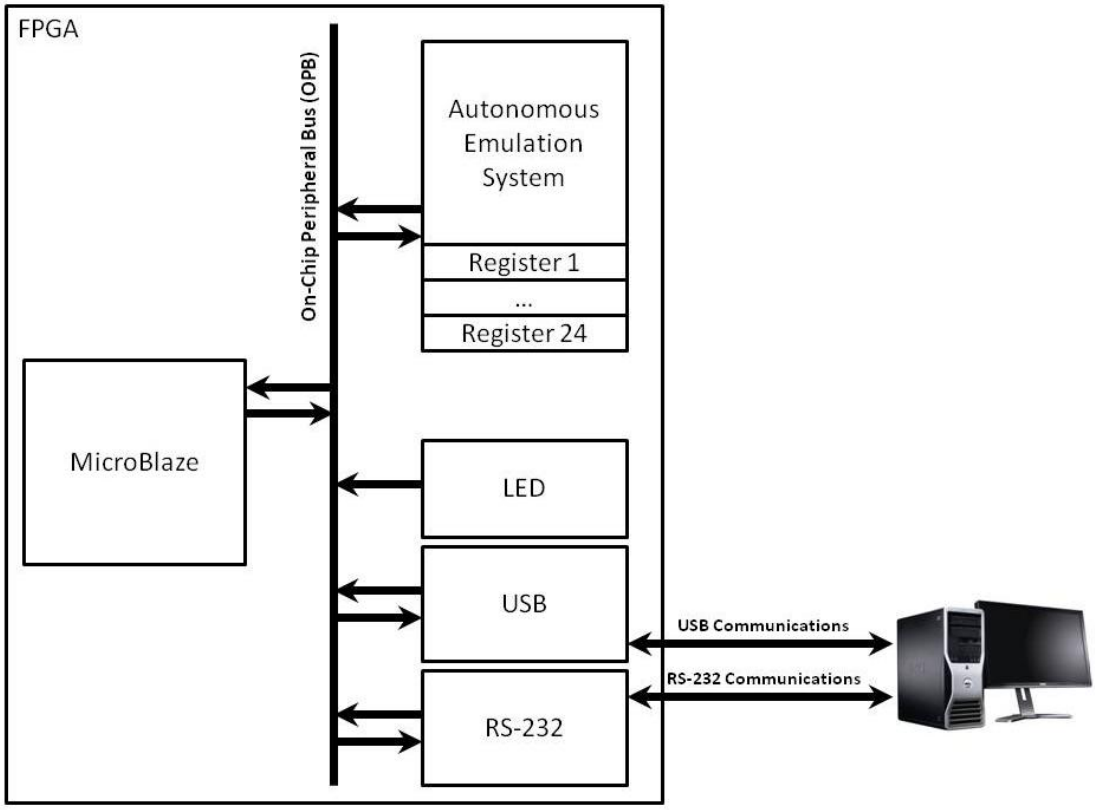

Figura 3.2: Arquitectura de un sistema de emulación autónoma con MicroBlaze.

El MicroBlaze es un microprocesador de 32 bits, su arquitectura está basada en conjunto de instrucciones reducidas, RISC, Reduced Instruction Set Computer, con 32 registros LUT de 32 bits, archivo de registro basado en RAM, con acceso a la memoria de datos e instrucciones separado.

Todos los periféricos están conectados a un bus, OPB (On-chip Peripheral Bus).

Para su inclusión en el sistema se ha empleado la herramienta de software anteriormente mencionada "Xilinx Platform Studio". Para crear el proyecto con el MicroBlaze se ha empleado el asistente para la creación de sistemas base, BSB (Base System Builder wizard) que posee la aplicación, lo que es muy recomendable.

A través del asistente, se han ido escogiendo los siguientes parámetros de configuración, que se ajustan a las necesidades del proyecto:

- Selección de tarjeta: el fabricante es AVNET, y la tarjeta es la Virtex 4 Evaluation Board (LX60).
- Microcontrolador: MicroBlaze.
- Frecuencia de reloj de referencia: 100 MHz.
- Frecuencia de reloj del bus del procesador: 100 MHz.
- Interfaz de depuración: depuración hardware en el chip.
- Memoria local: 64 KB de datos e instrucciones

3.3. PERIFÉRICOS EMPLEADOS

Continuando con el asistente, da la opción de seleccionar los periféricos que se quieren emplear. Para la creación de este proyecto se han empleado los siguientes periféricos:

- Comunicaciones RS-232: se utiliza el controlador OPB UART LITE, se ha configurado una tasa de transmisión de 115200 bits por segundo, y sin paridad.
- LEDs: se utiliza el controlador OPB GPIO, General Purpose Input Output, configurando los LEDs como salidas.
- Push_Button_SW_4: se utiliza el controlador OPB GPIO, configurando los pulsadores como entradas.
- DIP_Switches: se utiliza el controlador OPB GPIO, configurando los interruptores como entradas.
- FLASH_4Mx16: se utiliza el controlador OPB EMC, un controlador de memoria externa.
- Banco de 24 registros para comunicación entre el sistema de emulación autónoma y el MicroBlaze.

Una vez terminado el asistente es necesario añadir dos periféricos más:

- Controlador USB: se utiliza el controlador OPB USB, con una profundidad de memoria FIFO de 2048 bits, y un ancho de datos de 16 bits.
- Sistema de emulación de fallos: se utiliza un controlador creado mediante el asistente para la creación o importación de periféricos, conectado al bus OPB, y que está formado por 24 registros accesibles a través de software, de manera que estos registros serán el medio de comunicación entre el MicroBlaze y el sistema de emulación autónoma.

A continuación se exponen las particularidades de los controladores empleados para manejar los periféricos.

Controlador OPB UART LITE :

- Conectado con una interfaz OPB v2.0 de bus, con soporte de habilitación de byte.
- Soporta interfaces de bus de 8 bit, tiene un canal para recepción y otro canal para transmisión.
- Tiene una FIFO de 16 caracteres para transmisión y otra FIFO para recepción de 16 caracteres.
- El número de bits de datos es configurable entre 5 y 8 bits.
- Se puede configurar paridad par ó impar.
- Se puede configurar la tasa de transmisión.

Controlador OPB GPIO:

- Conectado con una interfaz OPB v2.0 de bus, con soporte de habilitación de byte.

- Configurable como canales GPIO sencillos o dobles.
- El número de bits GPIO se puede configurar de 1 a 32 bits.
- Cada bit GPIO es programable dinámicamente como entrada ó salida.
- Se pueden configurar como entradas para reducir la utilización de recursos.
- Puertos para conexiones tri-estado o no tri-estado.
- Valores de reset independientes para cada bit de todos los registros.
- Generación de petición de interrupción opcional.

Controlador OPB EMC:

- Parametrizado para un total de cuatro bancos de memoria (Síncrona ó Asíncrona).
- Direcciones base y rango de direcciones separado para cada banco de memoria.
- Conectado con una interfaz OPB v2.0 de bus, con soporte de habilitación de byte. El ancho de los datos de la memoria es independiente del ancho del ancho de los datos del bus OPB. Soporta anchos de 8 bits, 16 bits ó 32 bits.
- Se puede parametrizar el ancho de datos de la memoria y el ancho de datos del bus para que coincidan. Se ajustan múltiples ciclos de memoria cuando el ancho de la memoria es menor al ancho del bus OPB para una completa utilización del bus OPB. El ajuste del ancho de datos puede ser habilitado independientemente para cada banco de memoria.
- El tiempo de ciclo es configurable por parámetros de la hoja de características de la memoria.
- Soporta transmisiones de ráfaga o de bit a bit.
- Frecuencia de operación mayor o igual a 100 MHz.

Controlador OPB USB:

- Cumple con las especificaciones USB 2.0.
- Soporta los modos de comunicación High Speed y Full Speed.
- Tiene un interfaz con el bus OPB de 32 bits.
- Tiene un interfaz con estándar ULPI1 a USB PHY externo. Soporta relojes ULPI positivos y negativos.
- Tiene 8 endpoints, dentro de los que el 0 es el de control y del 1 al 7 pueden ser configurados en modo Bulk, Interrupt ó Isochronous. Los endpoints son configurables individualmente.
- Usa bloques de RAM para los buffers de los endpoints. Cada endpoint tiene dos buffers ping-pong.

3.4. COMUNICACIONES EMPLEADAS

Se han empleado tres tipos de distintos de comunicaciones: comunicaciones internas en la FPGA con el bus OPB, comunicaciones serie con protocolo RS-232 y protocolo USB 2.0, para las comunicaciones entre la FPGA y el MicroBlaze.

3.4.1. COMUNICACIÓN SERIE RS-232

Se ha utilizado comunicación serie a lo largo de toda la realización del proyecto con fines de depuración del software implementado en el MicroBlaze, dado que presenta una velocidad de transmisión inferior a las comunicaciones realizadas con el protocolo USB 2.0, pero es más estable la comunicación serie RS-232 y el software de comunicación en el ordenador no estaba en fase de desarrollo. Con lo que se ha utilizado de apoyo y para depuración.

En el ordenador se ha utilizado el programa TeraTermPro para recibir la información del MicroBlaze a través del COM1 del PC y para enviarle datos referentes a avances en los breakpoints.

En la siguiente tabla, la tabla 3.2, se muestran los parámetros empleados en la comunicación serie:

Puerto	COM1
Tasas de transmisión	115200
Datos	8 bits
Paridad	no
Bit de parada	1 bit
Control de flujo	no

Tabla 3.2: Parámetros de configuración del puerto RS232.

3.4.2. USB 2.0

La comunicación serie USB 2.0 consta de tres módulos: el controlador del MicroBlaze que está dentro de la FPGA, el microcontrolador de Cypress, CY7C68013, perteneciente a la tarjeta Virtex 4 Evaluation Board, y la aplicación desarrollada a lo largo de este proyecto en el ordenador.

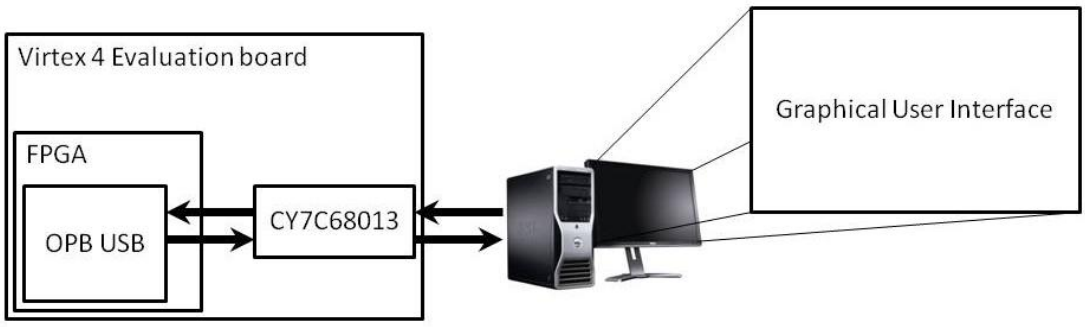

Figura 3.3: Esquema de comunicación USB.

En la Figura 3.3 se puede ver un esquema con los distintos módulos que intervienen en la comunicación USB.

Se ha utilizado el modo de transmisión Bulk, ya que este modo se utiliza para la transmisión de importantes cantidades de información. Al igual que el modo Control, este enlace no presenta pérdida de datos.

DESARROLLO DEL SOFTWARE PARA EL MICROBLAZE

4. DESARROLLO DEL SOFTWARE PARA MICROBLAZE

A continuación se explica las necesidades que se deben cumplir para poder realizar el nuevo sistema de emulación de fallos desde el punto de vista de la programación del microntrolador embebido MicroBlaze. Planteando los requisitos que debe cumplir el microntrolador y las medidas que se han llevado a cabo. Un sistema con un comportamiento igual al comportamiento del sistema que se parte, incluyendo la funcionalidad de un canal de comunicaciones USB.

4.1.ESPECIFICACIONES DE SOFTWARE

Para que el usuario controlase la carga de parámetros, el inicio de la emulación y la adquisición de los resultados, se utilizaba comunicación serie, a través del puerto serie del ordenador, con un programa de comunicaciones como el Teraterm ó el HyperTerminal . Con este Proyecto Fin de Carrera se pretende que el usuario sea capaz de controlar el sistema de Emulación Autónoma a través de una interfaz de comunicaciones que se comunique a través del puerto USB del ordenador.

Como ya se ha comentado en el capítulo anterior, el componente que se encarga de la transmisión de los parámetros y de dar la orden de arranque al Emulador es el MB. Para ello debe ser capaz de recoger los comandos que le son enviados desde el ordenador por USB y también debe ser capaz de enviar los resultados de la emulación de manera que la interfaz gráfica pueda gestionar estos datos, para posteriormente hacer un tratamiento de los mismos y poder presentarlos correctamente.

A lo largo de la realización del Proyecto Fin de Carrera se han usado las comunicaciones con el puerto serie para la depuración. Estas comunicaciones han sido en todo momento unilaterales, desde el MB hacia el ordenador.

Las comunicaciones con el protocolo de comunicación serie USB se basan en el envío y la recepción de paquetes de la misma longitud, de manera que la interfaz gráfica sea capaz de probar el correcto estado de las comunicaciones, pueda enviar los parámetros de configuración de la emulación, que sea posible para el usuario iniciar la emulación y se reciban en el ordenador los resultados de la emulación en tiempo real, es decir, a medida que el sistema de emulación autónoma vaya clasificando los errores que se encuentra en el circuito analizado.

Por último, el tercer tipo de comunicaciones que se tienen que implementar en el MB son las comunicaciones que se deben llevar a cabo entre éste y el sistema de Emulación Autónoma. Como ya se comentó en el capítulo anterior, para comunicar el MB con el Emulador se han usado 24 registros colgados del bus OPB. Estos registros se emplean para facilitar los parámetros y para dar la orden de inicio de la emulación.

Las especificaciones del software son:

- Realizar una prueba de comunicación USB.
- Recibir los parámetros de la emulación y transmitírselos al Emulador.
- Enviar la orden de inicio de emulación.
- Enviar los resultados de la emulación al PC.

Para cumplir con estas especificaciones se ha programado el MB de manera que realice unas configuraciones iniciales: crear variables a usar e inicializarlas, inicialización del emulador (se le envía un comando de reset a través del bus OPB), envío de un mensaje a través del puerto serie del ordenador para que el usuario compruebe que el MB está bien configurado y el programa está corriendo con normalidad, inicialización de la memoria cache del MicroBlaze, habilitación de las interrupciones que pudiese realizar el Emulador, informar mediante la UART que está listo para recibir comandos a través de USB.

Una vez realizadas las configuraciones iniciales, el MB ejecutará un bucle infinito, dentro del que, presenta una configuración de la comunicación USB en modo recepción de datos a través del endpoint 2, y espera instrucciones del ordenador. Estas instrucciones son las siguientes:

- Prueba de comunicación USB.
- Iniciar la emulación.
- Recepción de los parámetros de la emulación.
- Puesta a cero los registros del emulador.
- Mostrar los últimos resultados de la emulación.

En la Figura 4.1 se muestra un diagrama de flujo con el proceso principal de ejecución del programa desarrollado para el MB.

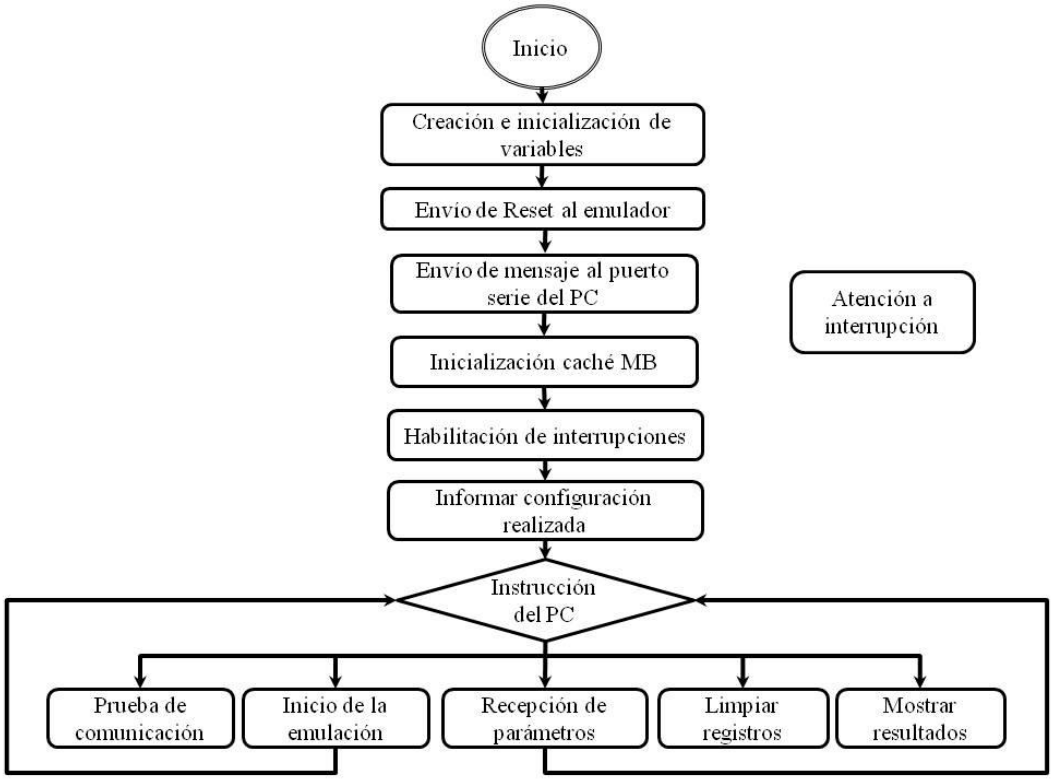

Figura 4.1: Diagrama de flujo del proceso principal del programa.

4.1.1. PRUEBA DE COMUNICACIÓN USB

La prueba de comunicación se basa en una respuesta del MB al ordenador, para probar que la comunicación USB está bien configurada. Informa a través del puerto RS-232 si el envío se ha realizado correctamente.

Para este propósito se emplea la función: USB_Test.

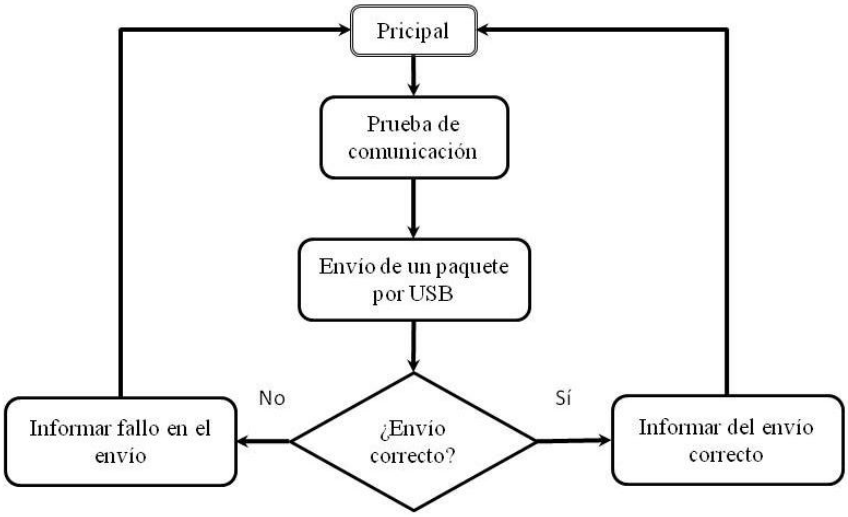

Figura 4.2: Diagrama de flujo del proceso de la prueba de comunicación.

4.1.2. INICIO DE EMULACIÓN Y GESTIÓN DE RESULTADOS

Al recibir el comando de inicio de la emulación, el MB almacena los resultados de la emulación en la variable registros y posteriormente los envía a través de USB. En primeras versiones se enviaba además la misma información a través de la UART, para cotejar los datos recibidos mediante USB y mediante RS-232. En la versión definitiva esto no se hace, ya que entonces se podrían perder datos al ser la comunicación por puerto serie más lenta que la USB y no valdrían los resultados obtenidos de la emulación.

Para el inicio de la emulación y la gestión de resultados se utilizan las funciones: inicia_emulacion, ResultadosDeEmulacion y usb_tx_results.

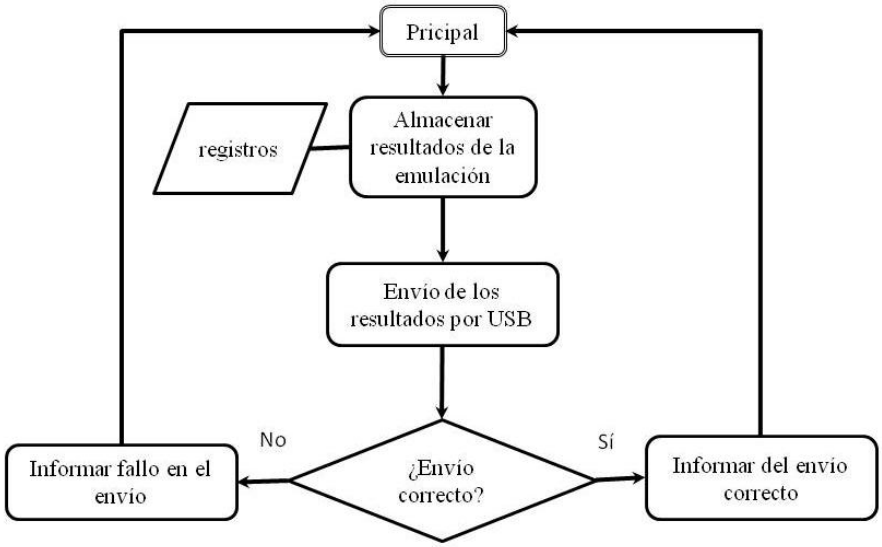

Figura 4.3: Diagrama de flujo del proceso del inicio de la emulación y gestión de resultados.

4.1.3. GESTIÓN DE PARÁMETROS DE EMULACIÓN

Cuando se recibe el comando de recepción de parámetros, se continúa con la configuración de recepción de datos y se procede a almacenarlos en el "array" registros, para posteriormente enviárselos al MB a través de los registros creados para el propósito de la comunicación entre el MB y el Emulador. Posteriormente se realiza una lectura de los registros del emulador y se envían los datos leídos mediante comunicación serie RS-232 al puerto serie del PC, por si el usuario quiere comprobar que los parámetros han sido almacenados correctamente en el Emulador.

Para la gestión de parámetros de la emulación se emplean las siguientes funciones: usb_rx_param y LecturaDeRegistros.

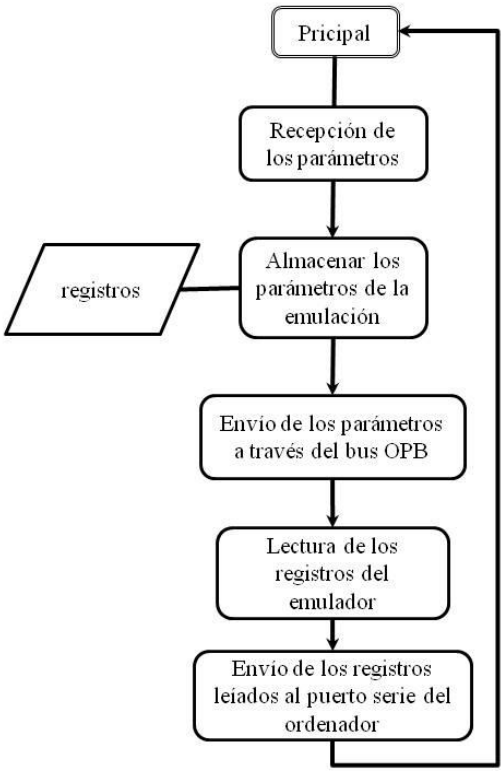

Figura 4.4: Diagrama de flujo del proceso de carga de parámetros en el emulador.

4.1.4. PUESTA A CERO DE LOS REGISTROS

Se resetea el valor de los registros del emulador, se envía los datos almacenados en los registros, los que deben valer cero todos, y se informa que la orden ha sido ejecutada con éxito. Esto se realiza mediante una escritura directa en los registros y para el envío de los datos almacenados en los registros se hace una llamada a las siguientes funciones: ResultadosDeEmulacion y usb_tx_results.

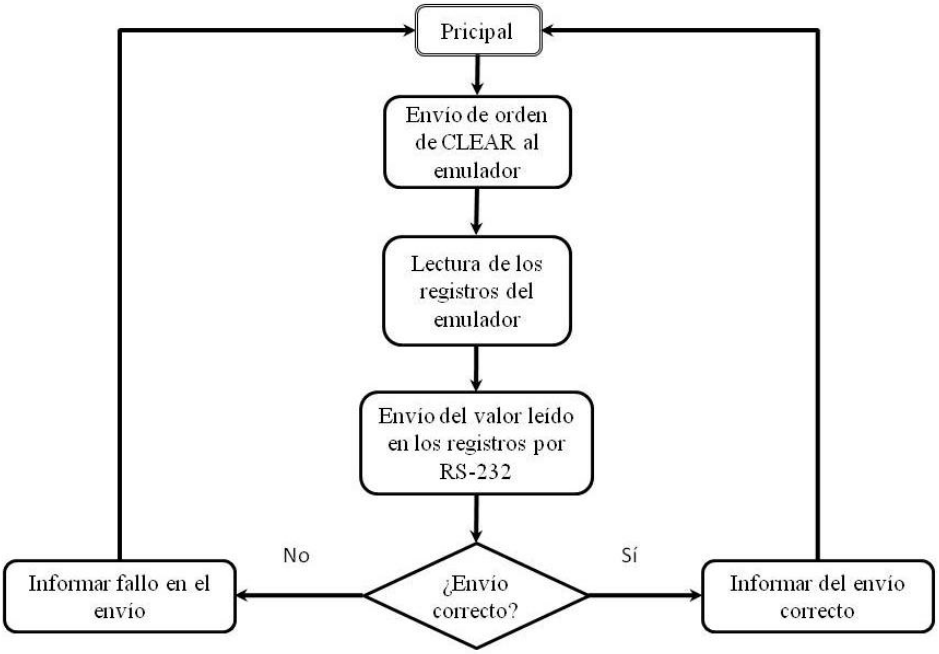

Figura 4.5: Diagrama del proceso de clear de los registros del emulador.

4.1.5. MOSTRAR LOS ÚLTIMOS RESULTADOS DE EMULACIÓN

Al igual que se hace cuando se ejecuta la instrucción de inicio de la emulación y gestión de resultados, se realiza una lectura de los registros y posteriormente se envían por USB al ordenador, pero previamente se envía la orden al emulador para que actualice sus registros. Para atender a esta instrucción se emplean las siguientes funciones: ResultadosDeEmulacion y usb_tx_results.

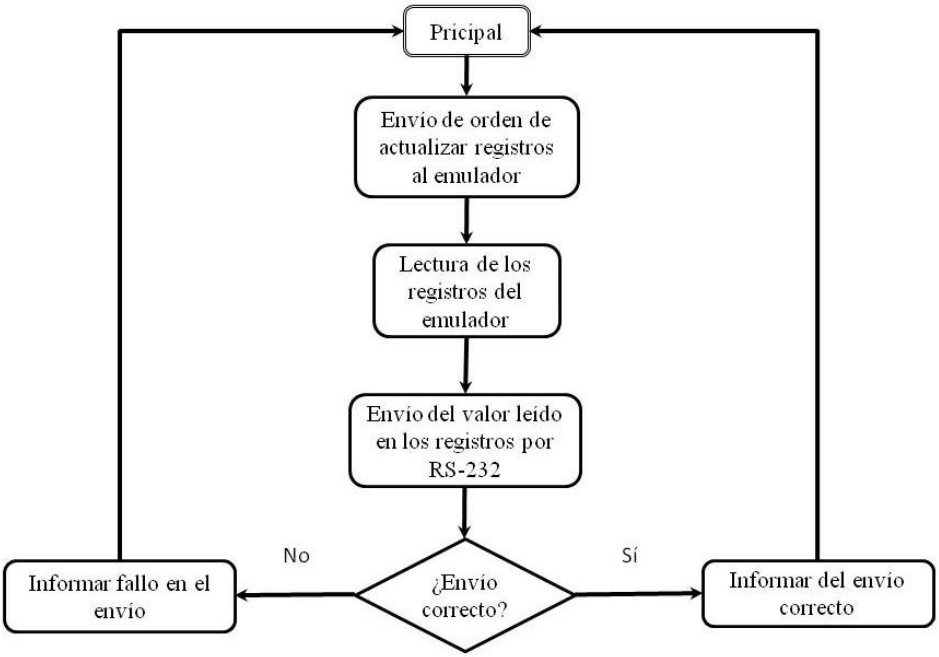

Figura 4.6: Diagrama del proceso de mostrar los últimos resultados de la emulación.

4.1.6. ATENCIÓN A INTERRUPCIÓN

Cuando el emulador provoca una interrupción MB ejecutará la rutina escrita en la función EMULADOR_Intr_DefaultHandler.

4.1.7. Función main

Como se muestra en la Figura 4.1, el flujo de la ejecución de la función principal, main(), comienza con la creación e inicialización de las variables que se van a emplear. Estas variables se describen en la Tabla 4.1.

Después se inicializa el emulador mediante la función "EMULADOR_mReset", para posteriormente informar que el MicroBlaze está correctamente inicializado, que este tipo de comunicación funciona correctamente, a través del puerto RS-232.

A continuación se configura la caché del MB, se configuran y habilitan las interrupciones que generará el emulador cuando tenga resultados nuevos y se informará de nuevo por puerto serie al ordenador que el MB está listo para recibir instrucciones a través del puerto USB.

A partir de este punto el MB estará ejecutando un bucle infinito en el que tendrá configuradas las comunicaciones USB en modo recepción, de manera que esperará que le lleguen instrucciones del ordenador a través del puerto USB.

Las instrucciones que atenderá son las mostradas en la tabla 4.2:

Instrucción	**Rutina**
0xFF	Enviar datos desde la FPGA al PC
0xFE	Recibir los parámetros de emulación
0xFD	Prueba de comunicación USB
0xFC	Limpiar registros del emulador
0xFB	Mostrar último resultado

Tabla 4.2: Relación instrucciones-rutinas.

4.2. DEPURACIÓN POR PUERTO SERIE

Para la depuración a través del puerto serie, se ha empleado la función "xil_printf", que es una función similar a "printf", pero con la peculiaridad de tener un tamaño mucho menor que esta última, solamente 1 KB. No soporta números en punto flotante, ni números "long long", por ejemplo números de 64 bits.

Este modo de depuración ha sido muy útil a la hora de depurar, sobre todo cuando aún no estaba implementada la comunicación USB, pero a medida que avanzaba la implementación también era de utilidad ya que se podían cotejar los datos enviados y recibidos en el MB. En la versión final se han dejado "breakpoints", para en un futuro poder saber cuál es el origen del fallo si el sistema da problemas, pero con volúmenes de transmisión menores.

Este tipo de transmisión, podría interferir en las comunicaciones a través del puerto USB, por lo que se ha puesto especial cuidado en no enviar tal cantidad de tramas que afecten al correcto funcionamiento del sistema.

En primer lugar se envía una trama para comprobar la correcta comunicación entre el ordenador y el MB, de manera que, si el MB está ejecutando el código correctamente y las comunicaciones serie a través del puerto serie están bien, se verá el siguiente mensaje: "Successful conection with Device Xilinx Virtex 4, XC4VLX60" en el Tera Term Web 3.1.

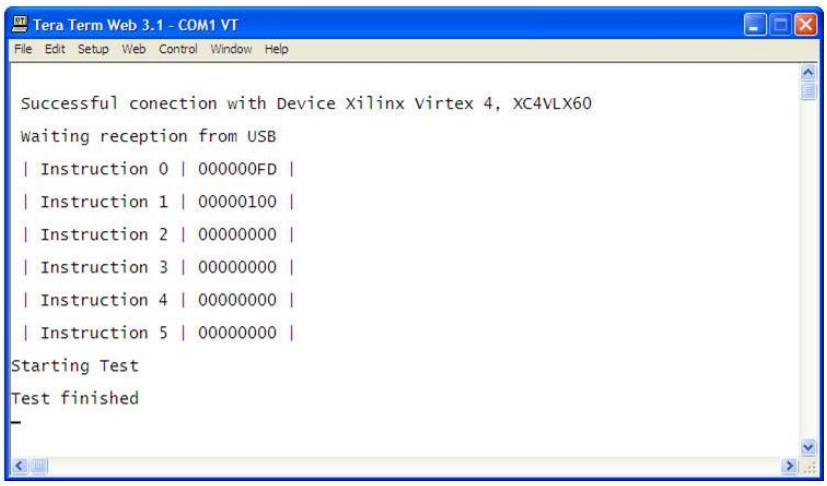

Figura 4.7: Comunicación serie con MB conectado y con instrucciones recibidas.

Cada vez que un comando sea recibido desde el ordenador a través de USB, se mostrarán las instrucciones recibidas por el MB, de manera que es posible saber lo que debería ejecutar el microcontrolador, como se puede ver en la Figura 4.8.

Cuando se reciban los parámetros de la emulación, se muestra un mensaje en el terminal informando que el MB está recibiendo los parámetros y posteriormente se mostrarán los valores almacenados en todos los registros empleados para la comunicación con el sistema de Emulación Autónoma, como se muestra en la Figura 4.8.

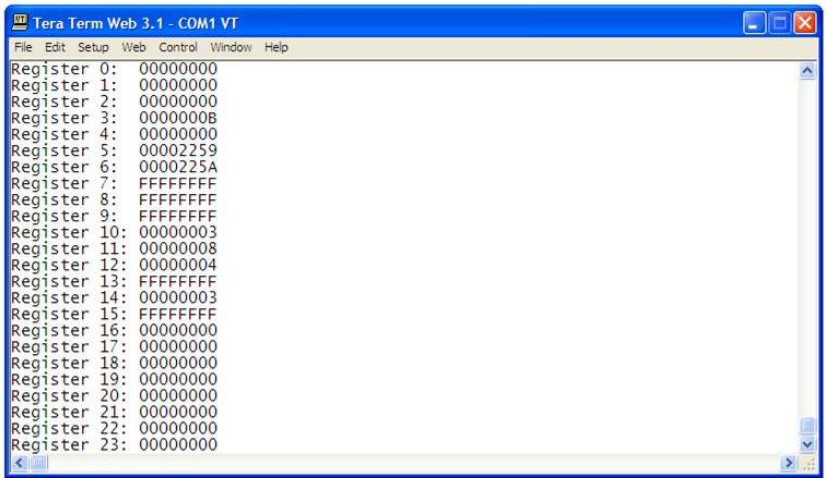

Figura 4.8: Comunicación serie con MB conectado y con instrucciones recibidas.

Además se envían alertas cuando:

- Se inicie la emulación.
- Finalice la emulación.
- Se haga una petición de prueba de la comunicación USB.

- Fallo en la transmisión de datos por USB.

Mientras se ejecuta la emulación, el MB irá enviando los resultados de la emulación a medida que los va generando el emulador, a través de USB. Además enviará al puerto serie del ordenador el valor de las instrucciones que recibe, en la instrucción número 3 se almacena el FF que se está probando en ese momento, y debajo de las instrucciones se muestra el FF de inicio y el FF de fin para dicha campaña de pruebas. De esta manera, en un punto medio de la emulación se verá en el terminal que el emulador está recibiendo la orden de iniciar emulación hasta que alcance el penúltimo FF, en este momento la interfaz del ordenador enviará la orden de mostrar resultados hasta que se hayan recibido los resultados del último FF que se pretendía probar.

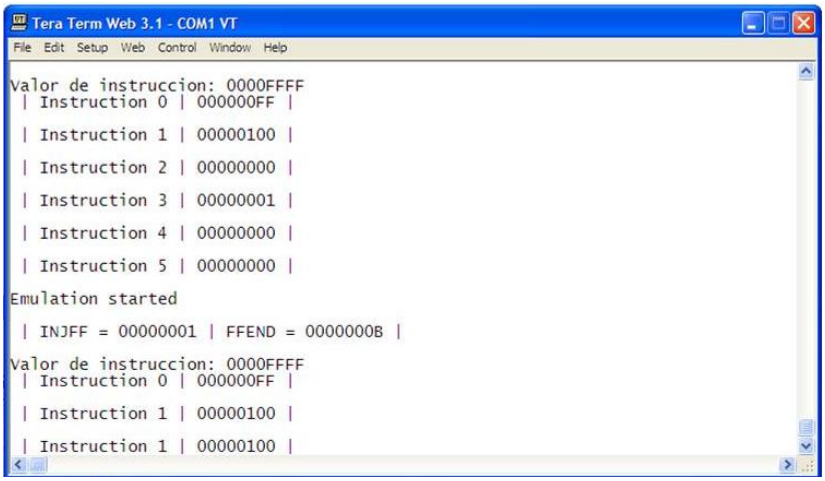

Figura 4.9: Comunicación serie con MB enviando resultados de la emulación.

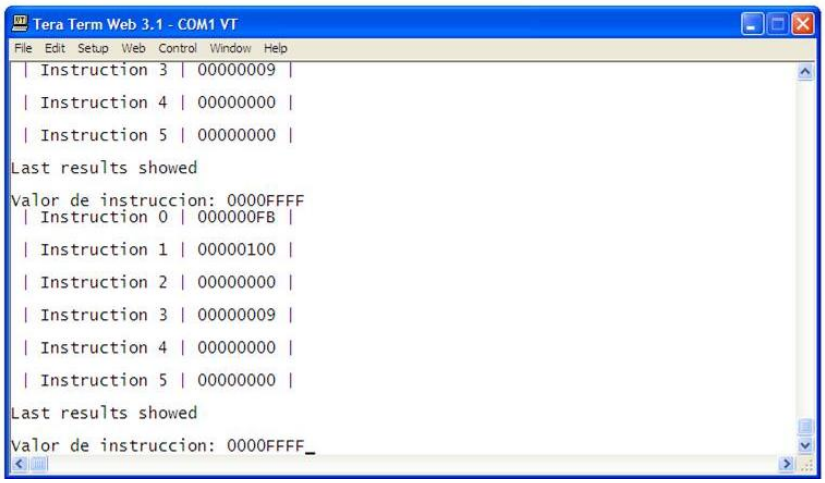

Figura 4.10: Comunicación serie con MB informando que la emulación ha concluido.

4.3. COMUNICACIÓN CON PC MEDIANTE USB

Todo el "software" que se ejecuta en el MB ha sido orientado principalmente para manejar las comunicaciones entre el ordenador y el MB a través del puerto USB del PC, para lo que se han implementado las siguientes funciones:

- usb_rx_param(Xuint8 endp_num, Xuint16 *instrucciones, Xuint16 *registros)

- usb_tx_results(Xuint8 endp_num, Xuint16 *instrucciones, Xuint16 *registros, Xuint8 FinEmulacion)
- USB_Test(Xuint8 endp_num)

4.3.1. Función usb_rx_param

Esta función es empleada para la recepción de los parámetros de emulación. Como parámetros recibe el "endpoint" a través del que se va a realizar la comunicación, el "endpoint" número 2, el puntero que apunta a la dirección de memoria dónde se han almacenado las instrucciones del recibidas del ordenador y por último el puntero que apunta a la dirección de memoria donde está almacenada la información que se quiere intercambiar con el sistema de Emulación Autónoma, es decir, los registros del emulador.

Se fija el endpoint en la dirección de lectura y se van almacenando todos los datos que envía el ordenador en la variable registros. Una vez obtenidos todos los datos, se llama a la función "ParametrosDeEmulacion" para cargar los parámetros en los registros del emulador.

4.3.2. Función usb_tx_results

Esta función es la encargada de enviar los paquetes con la información almacenada en los registros del emulador al ordenador. Al igual que la función "usb_tx_results" tiene como parámetros el "endpoint" a través del que se va a realizar la comunicación, en este caso el "endpoint" número 6, el puntero que apunta a la dirección de memoria dónde se han almacenado las instrucciones del recibidas del ordenador y por último el puntero que apunta a la dirección de memoria donde está almacenada la información que se quiere intercambiar con el sistema de Emulación Autónoma, es decir, los registros del emulador.

Se configura el endpoint en la dirección de transmisión de datos y se envía, en primer lugar, el contenido de la variable "Continuar", si esta variable tiene el valor 0xFFFF significa que la emulación aún no ha concluido, si por el contrario toma el valor 0x0000, significa que la emulación ha terminado y son los penúltimos resultados que deben enviarse al ordenador. En segundo lugar, se envía el contenido de la variable "NuevoFF", una variable global en la que se almacena el valor del FF que se está analizando en ese momento. Los demás datos que se envían son los almacenados en la variable registros, que son los valores que el emulador ha almacenado como resultados del FF analizado.

4.3.3. Función USB_Test

Cuando se realiza una prueba de comunicación USB, se envía un paquete desde el ordenador al MB, por lo que esta función se encarga de responder la petición de prueba. Para ello el único parámetro necesario es el "endpoint", que para el envío desde el MB es el número 6.

Se configura el endpoint en la dirección de transmisión de datos y se envían 200 bytes en respuesta a la petición de prueba de comunicación USB.

4.4. COMUNICACIÓN CON EMULADOR

Para realizar la comunicación entre el MicroBlaze y el sistema de Emulación Autónoma se ha creado un puente que está compuesto por un banco de 24 registros, que hacen la función de buffer de entrada/salida para dichas comunicaciones. A este buffer acceden tanto el emulador como el MicroBlaze, pero está integrado en la parte de VHDL, de Hardware Descrito. El MB, la

parte de software accede a los registros a través del bus OPB, mientras que el emulador escribe o lee en estos registros directamente.

En el desarrollo del software que ejecuta el MB se han implementado, para implementar la comunicación entre el MB y el emulador, las siguientes funciones:

- inicia_emulacion(void)
- ParametrosDeEmulacion(Xuint16 *registros)
- ResultadosDeEmulacion(Xuint16 *registros)
- LecturaDeRegistros(void)
- EMULADOR_EnableInterrupt(void * baseaddr_p)
- EMULADOR_Intr_DefaultHandler(void * baseaddr_p)

4.4.1. Función inicia_emulacion

Esta función es empleada para enviar la orden al emulador de inicio de la emulación. Esta función no recibe parámetros.

Se utiliza el registro “RCONTROL” para escribir en él la orden de iniciar emulación.

4.4.2. Función ParametrosDeEmulacion

Con “ParametrosDeEmulacion” se envían los parámetros de la emulación al sistema de Emulación Autónoma utilzando el bus OPB para escribir en los registros donde se deben almacenar los parámetros de la emulación. Recibe como parámetro la dirección de memoria de la variable registros.

Se escriben todos los registros del emulador con sus correspondientes valores, los que han sido recibidos y almacenados previamente a través del puerto USB.

Se utiliza una variable temporal, “temp”, de 32 bits para almacenar el valor que hay en el registro a leer, para posteriormente convertirla y almacenarla en la variable registro que es de 16 bits. Esto se hace, porque las transmisiones USB se realizan enviando 16 bits de datos.

4.4.3. Función ResultadosDeEmulacion

Con esta función se realiza una lectura de los registros del emulador, con esta función sólo se leen los registros que contienen los resultados de la emulación. Al igual que la función anterior recibe solamente como parámetro la dirección de memoria donde se almacena la variable registros.

Se almacenan los valores guardados en los registros “RCONTROL”, “RSTATUS”, “INJFF”, “SILENTS”, “FAILURES”, “LATENTS”, “RAMLATENTS”, “RUNCOUNT1” y “RUNCOUNT0”. “RCONTROL” y “RSTATUS” se leen para poder saber la orden que está ejecutando el emulador y el estado que presenta en el momento de la lectura, el resto de registros contienen los resultados de la emulación en el momento de la petición de resultados.

4.4.4. Función LecturaDeRegistros

Con "LecturaDeRegistros" se realiza una lectura de todos los registros del emulador. Al igual que las dos funciones anteriores, el único parámetro utilizado por esta función es la dirección de memoria donde se almacena la variable registros.

Es igual que la anterior, pero con dos diferencias, se leen todos los registros del emulador y además se envía el contenido de los mismos a través de la UART al puerto serie del ordenador.

4.4.5. Función EMULADOR_EnableInterrupt

Función para realizar la habilitación de las interrupciones que genere el sistema de Emulación Autónoma. Recibe como parámetro el punto del programa donde se quedó el MicroBlaze antes de realizar la atención a la interrupción.

Está función habilita las interrupciones provocadas por el emulador.

4.4.6. Función EMULADOR_Intr_DefaultHandler

Esta función es la función de atención a la interrupción generada por el emulador cuando tiene resultados listos cargados en los registros. Recibe como parámetro el punto del programa donde se quedó el MicroBlaze antes de realizar la atención a la interrupción.

La única instrucción que se ejecuta en esta función es incrementar en uno una variable global declarada llamada "EmulatorExpired". Está variable es comparada con otra propia de la función "main", si son distintas es que ha ocurrido una interrupción y el MB procederá a enviar los resultados nuevos de la emulación que está almacenados en los registros, cuando el ordenador se los haya solicitado a través de USB.

DESARROLLO DE LA INTERFAZ GRÁFICA DE USUARIO

5. DESARROLLO DE LA INTERFAZ GRÁFICA DE USUARIO

Dado que no existen aplicaciones similares a los terminales de comunicaciones serie para USB ha sido necesario implementar una aplicación de interfaz gráfica a medida para que la tarjeta con el sistema de emulación autónoma sea capaz de comunicarse con el PC. En este capítulo se explican las necesidades que se requieren y las mejoras que se han incorporado, dado que al hacer la aplicación a medida, ha sido posible incluir funcionalidades prácticas para un análisis en tiempo real de los resultados obtenidos en las emulaciones realizadas.

5.1. ESPECIFICACIONES DE LA APLICACIÓN

Los objetivos que se pretende conseguir con esta aplicación de interfaz gráfica de usuario, son, en primer lugar que exista una comunicación USB entre el ordenador y la tarjeta "Virtex 4 Development Board", y además que se pueda controlar el sistema de Emulación Autónoma desde el ordenador, siendo capaz la aplicación de almacenar los resultados de la emulación y realizar un tratamiento de los datos obtenidos.

La aplicación está divida en dos pestañas. La primera es de configuración, en la que se elige la tarjeta que se conectará por USB al ordenador. Al realizar la selección de la tarjeta se cargarán los parámetros propios de dicha tarjeta. Además en esta pestaña se podrán elegir las direcciones de los archivos utilizados por la aplicación.

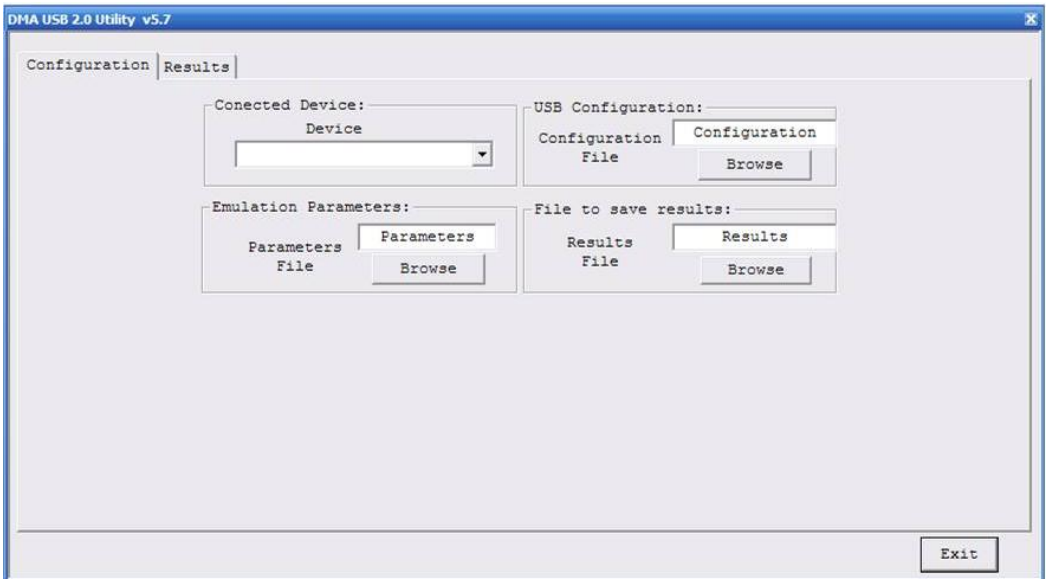

Figura 5.1: Pestaña de configuración.

Por otro lado se ha implementado una pestaña que actuará de interfaz entre la tarjeta y el usuario. En esta pestaña se podrán enviar órdenes al MicroBlaze a través del USB y se podrán ver los resultados de la emulación en tiempo real a medida que vayan llegando los resultados al ordenador.

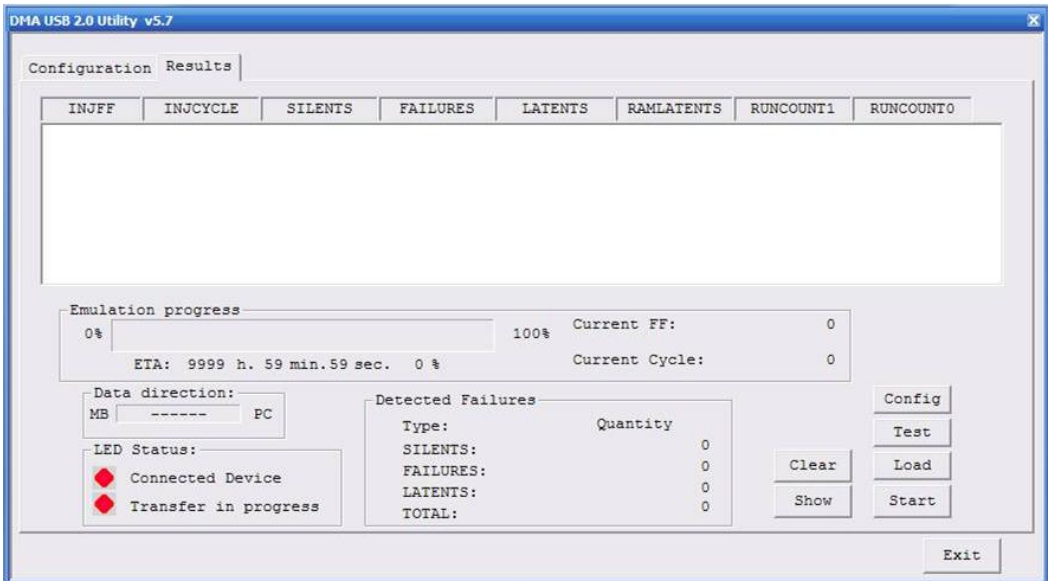

Figura 5.2: Pestaña de resultados.

5.2. CONFIGURACIÓN

Esta pestaña consta de cuatro elementos. En primer lugar presenta un desplegable mediante el que se puede escoger la tarjeta que se va a conectar a través de USB al ordenador. Esta elección implica escoger los parámetros de configuración propios de la tarjeta. En el diseño de la aplicación se han tenido en cuenta sólo dos tarjetas, la "Virtex 4 Development Board" y la "Virtex 5 Xilinx University Program". Además de estas dos opciones, hay una tercera posible elección "Generic", con la que se interpreta que hay conectada una tarjeta genérica similar a la "Virtex 4 Development Board" y se cargarán los parámetros de esta tarjeta.

Por otro lado hay tres controles en los que se les puede indicar a la aplicación la ruta donde están los archivos que la aplicación utilizará. Los archivos que esta aplicación utiliza son:

- El archivo de configuración de la comunicación USB, con el que se programará el microcontrolador, en el caso de elegir las opciones "Virtex 4 Development Board" ó "Generic" en la pestaña desplegada explicada en el párrafo anterior se tratará del microcontrolador de Cypress, CY7C68013.

- El archivo que contiene los parámetros de la emulación que se le enviarán al MicroBlaze para que éste se los envíe al Emulador.

- El archivo donde se almacenarán los resultados de la emulación.

Por defecto se utilizarán unos valores que implican trabajar con los archivos en la misma carpeta donde se ejecuta la aplicación, pero con esta pestaña se podrían modificar. El archivo donde se almacenan los valores por defecto es el archivo "Configuration.ini" que debe estar en la misma carpeta donde se ejecuta el programa de interfaz.

5.2.1. SELECCIÓN DE TARJETA

Por defecto se utilizarán unos valores que implican trabajar con los archivos en la misma carpeta donde se ejecuta la aplicación, pero con esta pestaña se podrían modificar. El archivo

donde se almacenan los valores por defecto es el archivo "Configuration.ini" que debe estar en la misma carpeta donde se ejecuta el programa de interfaz.

Cuando se escoge una de las opciones del desplegable, la aplicación cargará el archivo Configuration.cfg, para después reconocer qué opción se ha escogido, y a consecuencia de esta elección, cargar en la variable global "m_intBoardIDAddr" el identificador de tarjeta conectada. Posteriormente llamará a la función "UpdateConfig" dentro de la que se actualizará el archivo de configuración de la aplicación.

La función "UpdateConfig" se encarga de actualizar el archivo de configuración de la aplicación con valores por defecto, ó bien, con los valores seleccionados por el usuario. En la Tabla 5.1 se puede observar un resúmen de las variables empleadas.

Primero se lee todo el archivo haciendo una copia temporal de su contenido. Posteriormente se realiza una escritura del mismo, dónde se le incluirán los datos necesarios, el tipo de tarjeta conectada y las rutas de los archivos que utiliza la aplicación para la Emulación, en el caso que no se hayan indicado previamente la ruta de los archivos, se cargará unos valores por defecto.

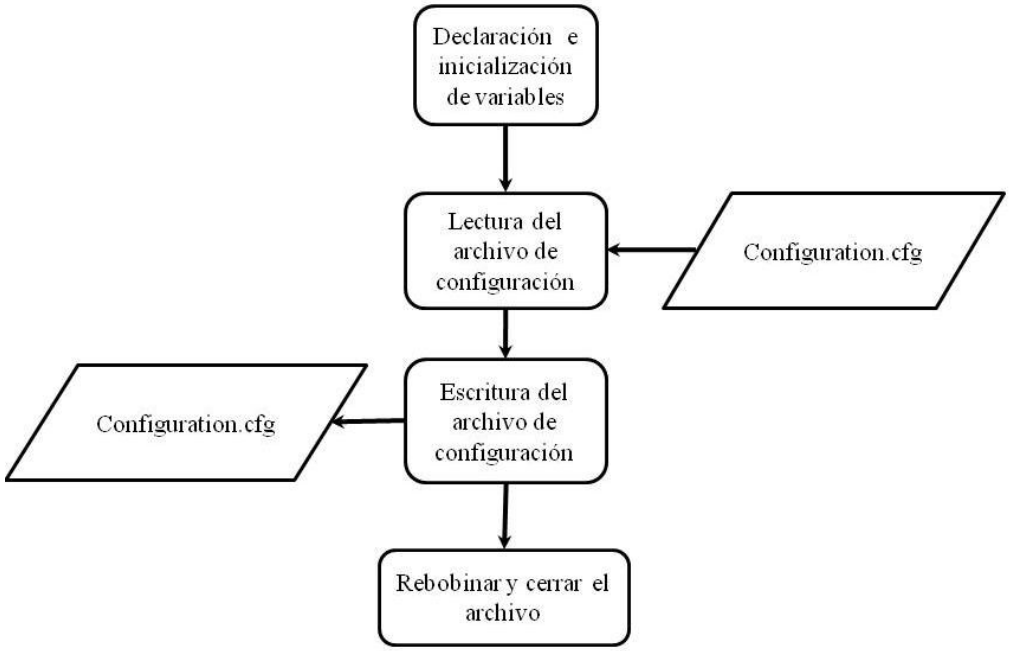

Figura 5.3: Diagrama de flujo de la función UpdateConfig.

5.2.2. RUTA DE ARCHIVO

Para la modificación del archivo de configuración se deben seleccionar las rutas de los tres archivos y en último lugar seleccionar la tarjeta en el menú desplegable. Cuando se pulsa en cualquiera de los tres botones de selección de archivo, se realiza una llamada a la función "GetFileName", dependiendo del botón que se haya pulsado en la ventana emergente de Windows para la selección de archivo se indicará el tipo de archivo esperado y posteriormente se almacenará en las variables globales m_strCntlFilename y m_strPathName, el nombre y la ruta del archivo.

Posteriormente se volverá a la función de atención a la pulsación del botón y en esta función se almacenará en sus correspondientes variables globales el nombre y la ruta del archivo.

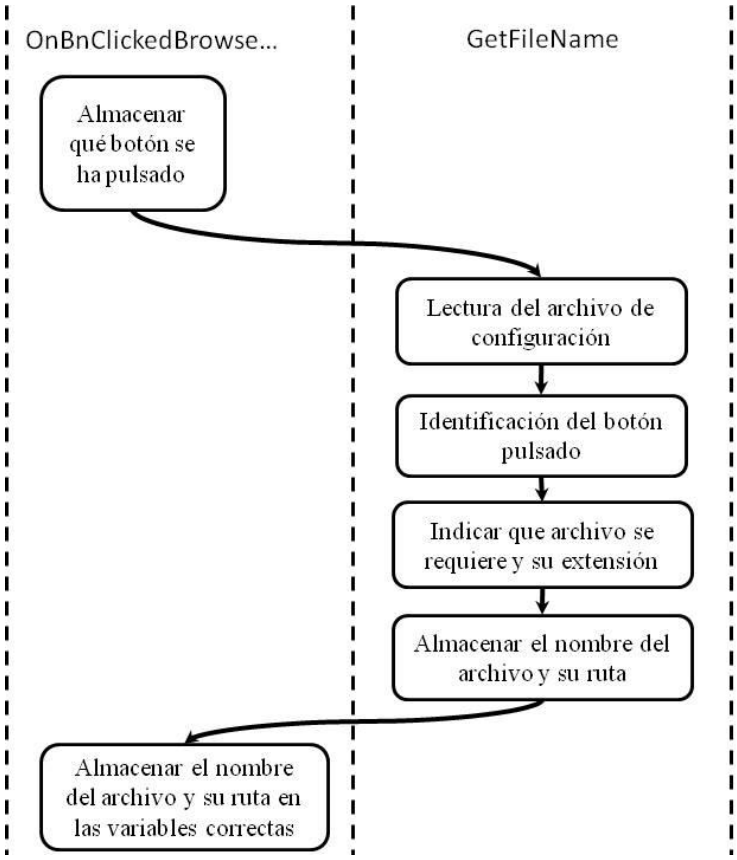

Figura 5.4: Diagrama de flujo para determinar la ruta de los archivos.

5.3. COMUNICACIÓN POR USB

Para implementar la comunicación USB se ha partido de una configuración en la que el "host", el ordenador, actúa como maestro del bus mientras que el periférico conectado, es decir, la tarjeta funciona como esclava. Esto implica que quién inicia las comunicaciones es la aplicación y espera respuesta dependiendo de lo que se está ejecutando. Por su parte la tarjeta conectada deberá estar en todo momento esperando recepción de órdenes y actuar consecuentemente en el momento que las reciba.

Las comunicaciones USB que se plantean son cuatro:

- Configuración del microcontrolador usado para la comunicación USB, en esta comunicación la tarjeta recibirá la orden de configuración del dispositivo y posteriormente el archivo de configuración.

- Prueba de las comunicaciones USB, esta prueba consiste en el envío desde el ordenador a la tarjeta de un paquete, y después la tarjeta enviará otro paquete al ordenador. Si el ordenador recibe el paquete de prueba se da por concluido el test y se considerará como correcto el funcionamiento de las comunicaciones USB.

- Envío de los parámetros de emulación. Mediante está transmisión la aplicación irá enviando los parámetros de configuración para el Emulador. Una vez cargados los parámetros de emulación en el sistema de Emulación Autónoma la aplicación esperará a recibir el contenido de los registros, estos datos se almacenarán en un archivo de

registro por si el usuario desea conocer si los parámetros han sido correctamente recibidos y almacenados.

- Envió de la orden de inicio de emulación. Una vez enviada la orden de inicio de emulación, la aplicación irá pidiendo resultados y recibiéndolos hasta el fin de la emulación. Durante este proceso la aplicación no aceptará ningún tipo de interacción con el usuario, dado que cualquier atención a instrucciones del usuario podría conllevar la pérdida de resultados, lo que no es deseable y es uno de los motivos por los que se implementó la comunicación USB entre el ordenador y la tarjeta en la emulación de circuitos.

5.3.1. Función TestUSBComm

Con esta función se implementa un test de las comunicaciones USB entre el ordenador y la tarjeta conectada, para que, antes de empezar con la emulación se sepa si el canal de comunicaciones está correctamente configurado y listo para usarse.

Esta prueba consiste en realizar un envío al MicroBlaze y esperar su respuesta. Si no se consigue realizar el envío será porque el MB no responde con lo que habrá un problema en las comunicaciones por lo que el test no se dará por bueno. Igualmente si tras realizar un envío correcto no se obtiene respuesta se dará por incorrecto el test porque existe un problema en las comunicaciones PC-MB.

A continuación, en la Tabla 5.3, se exponen las variables empleadas por esta función que implementa el test de comunicación USB.

En primer lugar se declaran e inicializan las variables empleadas en la función. El siguiente paso es abrir el dispositivo para la transmisión y para la recepción, ya que con esta función se van a establecer comunicaciones bilaterales. Lo que se hará con la función bOpenDriver, a la que se le pasarán como parámetros el manipulador con la información del dispositivo empleado en el envío ó recepción de datos, en función del tipo de apertura (para envío ó para recepción), y el nombre del dispositivo almacenado en la variable DeviceName.

A continuación se fijarán los objetos de evento mediante la función CreateEvent. El valor que devuelva esta función será guardado en las variables WriteCompleteEvent y ReadCompleteEvent, con el propósito de tener unos manipuladores para poder comprobar si tanto la recepción como el envío han concluido correctamente.

El siguiente paso es fijar los valores de las tuberías a través de las que se van a transmitir datos y crear los buffers en los que se almacenarán temporalmente tanto los datos recibidos como los empleados. Para cargarle los comandos que se van a enviar por USB al MB.

Una vez inicializado, se procede al envío de los comandos, con los que se le indica al MicroBlaze que se va a realizar una prueba de comunicación. En la siguiente tabla se muestra los comandos que se envían por USB.

Buffer	**Valor**	**Información**
outBuffer[0]	TEST	
outBuffer[1]	0x00	Byte reservado para uso futuro
outBuffer[2]	0x00	Byte menos significativo del tamaño de la transmisión
outBuffer[3]	0x01	Byte más significativo del tamaño de la transmisión
outBuffer[4]	0x00	Byte reservado para uso futuro
outBuffer[5]	0x00	Byte reservado para uso futuro
outBuffer[6]	0x00	Byte reservado para uso futuro
outBuffer[7]	0x00	Byte reservado para uso futuro
outBuffer[8]	0x00	Byte reservado para uso futuro
outBuffer[9]	0x00	Byte reservado para uso futuro
outBuffer[10]	0x00	Byte reservado para uso futuro
outBuffer[11]	0x00	Byte reservado para uso futuro

Tabla 5.4: Comandos de los buffers de transmisión.

Ya sólo queda inicializar las estructuras de datos para la transmisión, iniciar el envío de los datos y esperar a que se termine de transmitir toda la información.

Las estructuras, tanto de transmisión como de recepción de datos para la comunicación USB son muy similares, sólo cambian los parámetros con los que se las inicializa. En la Tabla 5.5 se muestra, a la izquierda las estructuras de datos para la transmisión, y a la derecha, las estructuras de datos para la recepción. Como se aprecia en la siguiente tabla, las estructuras de datos son estructuras en las que se almacenan los datos de inicialización para la comunicación mediante el control OutBulkControl, para la transmisión, y para la recepción, InBulkControl. Estos controles tienen parámetros como el número de tubería por el que se va a comunicar, el manipulador, el tipo de transmisión USB que se va a establecer, la dirección de memoria donde se almacena el control, el tamaño del control, el buffer donde se han almacenado los datos a enviar, ó el buffer donde se almacenarán los datos recibidos, el tamaño de la transmisión ó recepción, y por último, el evento de escritura ó lectura correcta.

outBulkControl.pipeNum = OutPipeNum;
 	
outThreadControl.hDevice = hOutDevice;
 	
outThreadControl.Ioctl = IOCTL_EZUSB_BULK_WRITE;
 	
outThreadControl.InBuffer = (PVOID)&outBulkControl;
 	
outThreadControl.InBufferSize = sizeof(BULK_TRANSFER_CONTROL);
 	
outThreadControl.OutBuffer = outBuffer;
 	
outThreadControl.OutBufferSize = outXferSize;
 	
outThreadControl.completionEvent = WriteCompleteEvent;
 	
outThreadControl.status = FALSE;
 	
outThreadControl.BytesReturned = 0;
	inBulkControl.pipeNum = InPipeNum;
	
	inThreadControl.hDevice = hInDevice;
	
	inThreadControl.Ioctl = IOCTL_EZUSB_BULK_READ;
	
	inThreadControl.InBuffer = (PVOID)&inBulkControl;
	
	inThreadControl.InBufferSize = sizeof(BULK_TRANSFER_CONTROL);
	
	inThreadControl.OutBuffer = &inBuffer[0];
	
	inThreadControl.OutBufferSize = inXferSize;
	
	inThreadControl.completionEvent = ReadCompleteEvent;
	
	inThreadControl.status = FALSE;
	
	inThreadControl.BytesReturned = 0;

Tabla 5.5: Estructuras de datos para la comunicación USB.

En cuanto se complete la transmisión y no haya fallado el envío, se envía la petición de recepción de datos de una forma muy similar a la utilizada para transmitir.

Y si la recepción es correcta, la función liberará el espacio reservado para almacenar todos los datos que ha necesita para establecer la comunicación USB y devolverá un valor TRUE, informando que la prueba de comunicación ha sido correcta.

En el diagrama de flujo de la Figura 5.5 se puede ver de forma más general los pasos que sigue en su ejecución la función TestUSBComm.

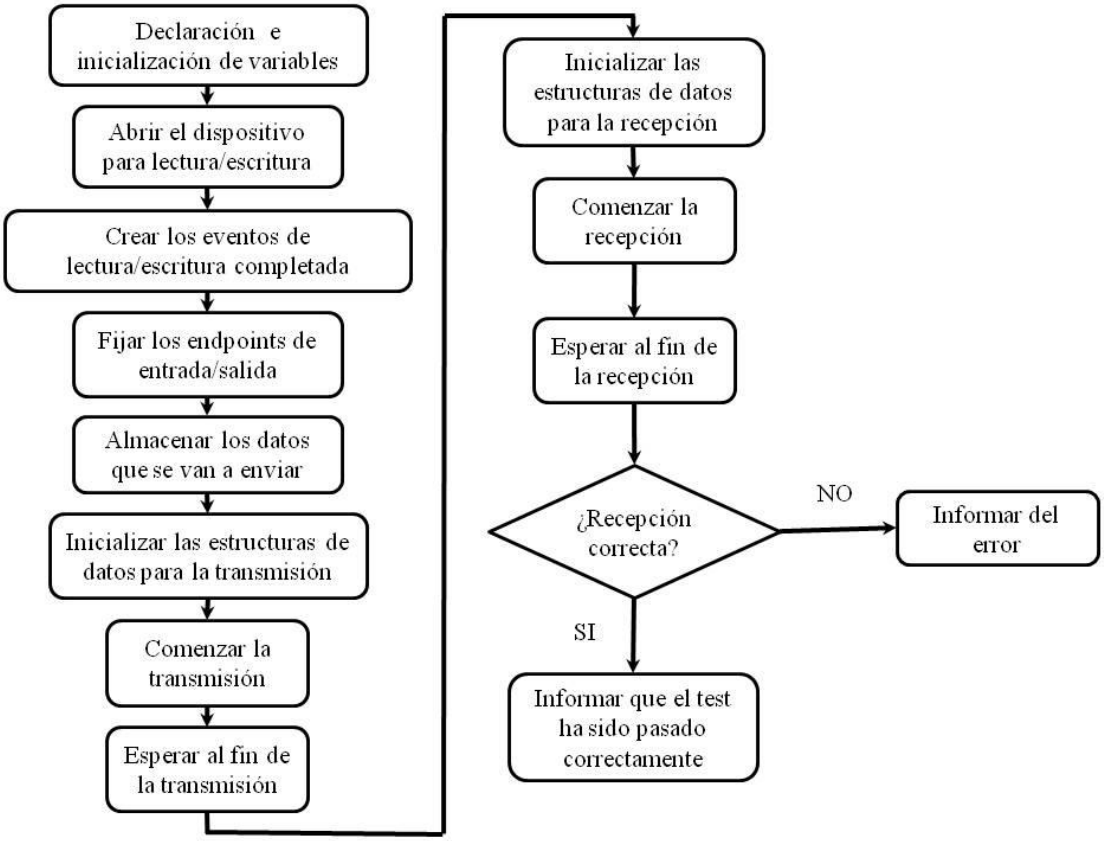

Figura 5.5: Diagrama de flujo de la función TestUSBComm.

5.3.2. Función ReadConfig

Con esta función la aplicación obtiene los parámetros necesarios para la establecer una correcta comunicación entre el PC y el MB mediante USB, lo que incluye la ruta de los archivos empleados y parámetros propios de la tarjeta a la que se conecta el ordenador mediante el cable de comunicaciones USB.

A continuación, en la Tabla 5.6, se exponen las variables empleadas por esta función que implementa la lectura de los datos de configuración empleados por la interfaz.

Tras declarar las variables que se van a emplear e inicializarlas, se realiza una lectura del archivo Configuration.cfg, el que contiene la información referente al tipo de tarjeta que estará conectada al ordenador, y las rutas de los archivos que la interfaz utilizará para poder realizar la emulación del circuito. Todos estos parámetros se almacenan para posteriormente ser empleadas por otras funciones de la aplicación.

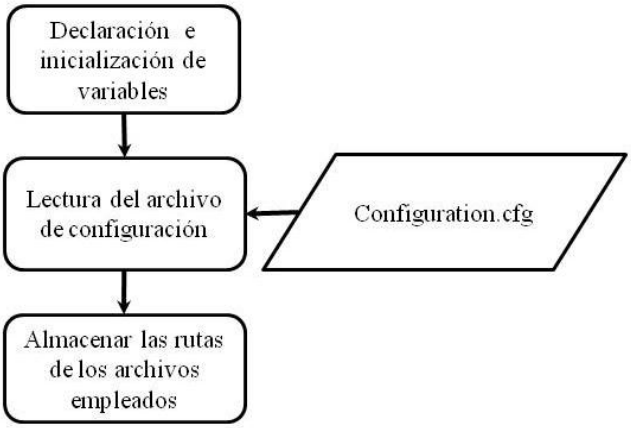

Figura 5.6: Diagrama de flujo de la función ReadConfig.

5.3.3. Función ConfigureUSB

Como ya se ha comentado anteriormente se emplea un microcontrolador que hará de intérprete en las comunicaciones USB. Esta función es la encargada de configurar dicho dispositivo para que se puedan realizar las comunicaciones USB correctamente.

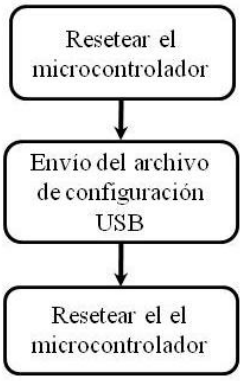

Figura 5.7: Diagrama de flujo para configurar la comunicación USB.

Para configurar el microcontrolador, primero se reseteará, y posteriormente se le enviará el archivo de configuración donde se incluyen todos los parámetros necesario para configurar dicho dispositivo con el modo correcto que se va a emplear en las comunicaciones y las direcciones de las tuberías, tanto la de transmisión como la de recepción. Una vez cargado el archivo de configuración en el microcontrolador, se le enviará una orden para que se resetee y esté listo para llevar a cabo las comunicaciones entre el PC y el MicroBlaze.

5.3.4. Función SendTrama

Esta función tiene como objetivo establecer una comunicación entre el PC y el MB, con el fin de enviar desde un archivo, los parámetros de emulación, almacenados en dicho fichero. Para ello, deberá enviarle al MicroBlaze el comando correspondiente, para que éste se configure en modo recepción de parámetros y pueda almacenarlos y transmitírselos correctamente al sistema de Emulación Autónoma.

A continuación, en la Tabla 5.7, se exponen las variables empleadas por esta función que implementa el envío de los parámetros de la emulación.

Como se puede ver en la Figura 5.8 tras la declaración e inicialización de las variables empleadas, se prepara el dispositivo para la comunicación USB. Para ello, se abre el dispositivo para escritura, se crean los eventos para determinar si la escritura se ha completado y se fija la tubería de comunicaciones de salida. En este momento se abre el archivo que contiene los parámetros de la emulación para realizar una copia temporal de los mismos. Se fija el tamaño de los paquetes que se van a transmitir, y se envía la orden correspondiente, que el MB interpretará de manera que espere los parámetros de emulación que le van a ser enviados a través de la comunicación USB.

Posteriormente se llena el buffer de salida con los datos a enviar y se inicializan las estructuras de datos para la transmisión. Se inicia la transmisión y la aplicación esperará a que se haya realizado correctamente. En el caso que la transmisión hubiese fallado se informará al usuario del error y se saldrá de la función. Si por el contrario, la transmisión es correcta, se informa al usuario que los parámetros de emulación han sido enviados correctamente al MicroBlaze.

Al final se liberará el espacio reservado por las variables declaradas como puntero y para las que se reservó espacio al inicio de la función y se cerrará el canal de comunicaciones creado.

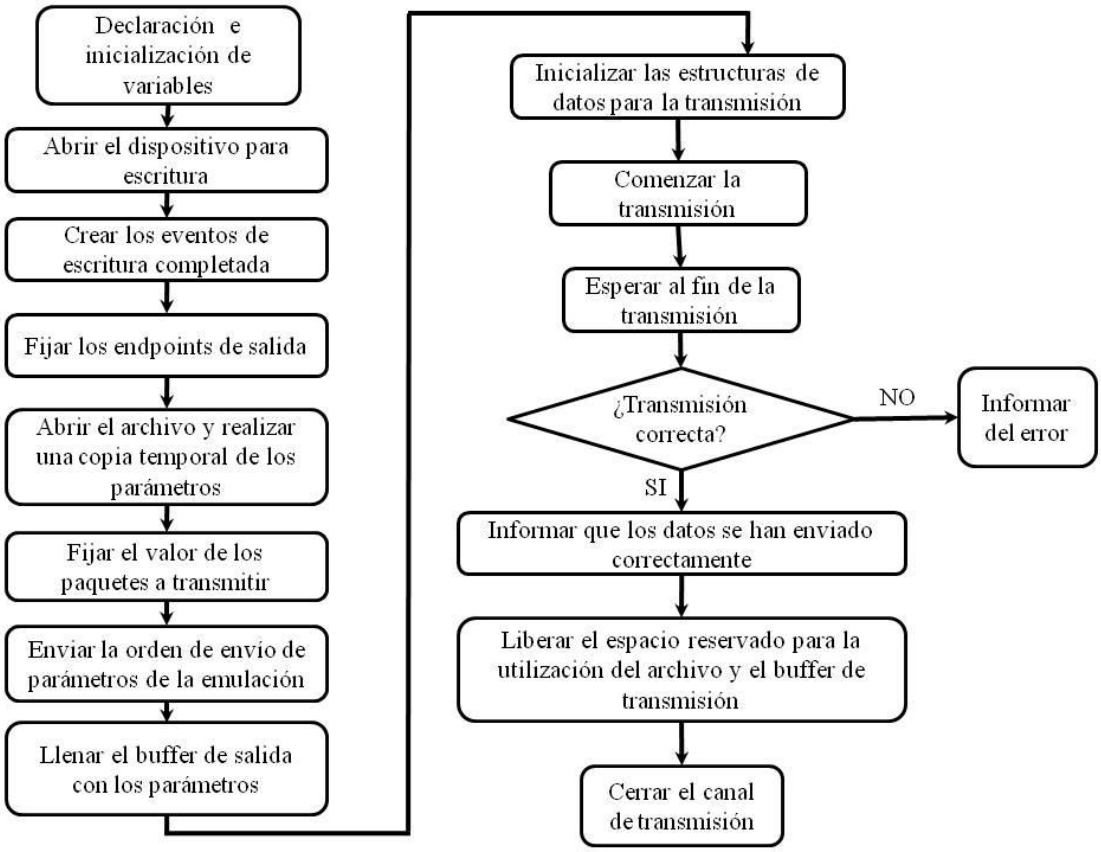

Figura 5.8: Diagrama de flujo para enviar los parámetros de la emulación.

5.3.5. Función ReadTrama

Todas las funciones que requieran realizar una petición de resultados de emulación, los que son enviados desde el MB al ordenador realizarán una llamada a esta función. La función

ReadTrama es la encarga de establecer un canal de comunicaciones con sentido MB al PC. Además esta función enviará el comando correspondiente al MB para que el dispositivo interprete que se requiere una transmisión de datos para ser usados por la interfaz.

Otra de las funcionalidades implementadas en ReadTrama es la de mostrar información referente al progreso de la emulación, para poder ser vistos por el usuario. Para ello se han implementado cálculos para poder facilitar información tal como, tiempo estimado para el fin de la emulación, progreso en tanto por ciento de la emulación y resultados actuales obtenidos.

A continuación, en la Tabla 5.8, se exponen las variables empleadas por esta función que implementa adquisición de los resultados.

Esta función implementa tanto transmisión como recepción. Transmisión empleada para enviar el comando correspondiente al MB y recepción para almacenar los resultados de emulación. En la Figura 5.9 se muestra un resumen de cómo se establecen las comunicaciones entre la aplicación y el MicroBlaze. Esta funcionalidad es similar a la explicada para la función SendTrama con la salvedad que en este caso se realizan comunicaciones bilaterales.

En este caso se abre el dispositivo tanto para lectura como para escritura. Se fijan los endpoints de entrada y salida y se inicializan los buffers empleados en la comunicación.

Como ya se ha comentado anteriormente, esta función es empleada por otras funciones, por este motivo, esta función recibe un comando con el que se indica qué función ha sido la que la ha llamado y cómo se debe comportar en función de dicho comando. Para ello en este punto de la función se identificará el comando recibido.

Una vez identificado el comando se inicializarán las estructuras de escritura, en la que se incluirá un comando, este comando tiene como destinatario el MB, el que interpretará para realizar la acción correspondiente.

Una vez listo todo lo anteriormente expuesto, se inicia la transmisión y se espera a que la transmisión concluya. Si ha fallado la transmisión se informará del error y, si por el contrario, la transmisión ha sido correcta se inicializarán las estructuras de datos para la recepción.

Una vez inicializadas las estructuras de datos para la recepción se inicia la comunicación entrante y se espera a que finalice. Cuando finalice la comunicación, si ha sido correcta se procesarán los datos recibidos, en caso contrario, se informa del error en la recepción al usuario.

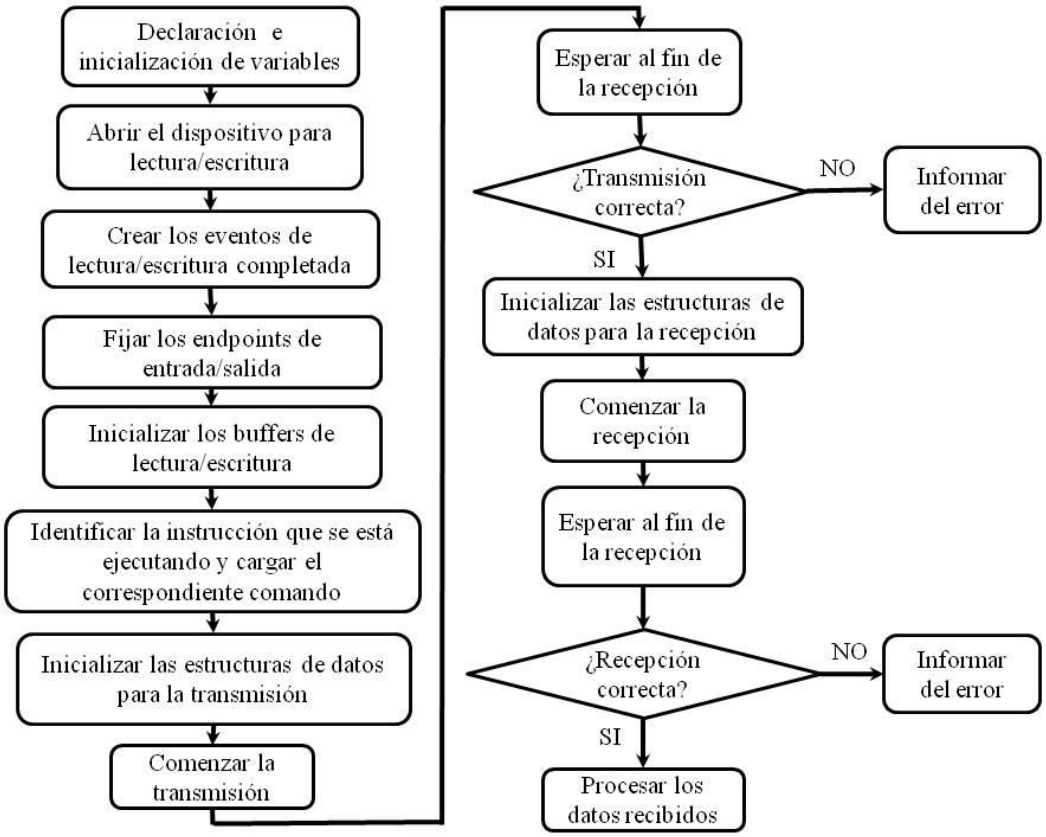

Figura 5.9: Diagrama de flujo para enviar un comando al MicroBlaze por USB.

Si los resultados han sido recibidos correctamente la función procede a realizar el tratamiento de los datos. Este tratamiento de datos depende del comando que haya recibido esta función. El comando puede ser uno de los siguientes: iniciar emulación, limpiar registros, mostrar resultados ó mostrar último resultado.

En el caso de haber recibido el comando que indica iniciar la emulación, la función realizará los pasos mostrados en el diagrama de flujo de la Figura 5.10.

Si la emulación no se ha iniciado aún, se inicializa el archivo de resultados incluyendo hora de inicio de la emulación y las cabeceras para poder identificar a qué resultado corresponde el valor mostrado.

Si la emulación ya había sido iniciada, es decir, es la segunda o sucesivas veces que se llama a esta función con el comando de iniciar emulación, en primer lugar se actualizará el archivo de resultados con los valores recibidos del MicroBlaze.

A continuación se realizan los cálculos necesarios para obtener el tiempo estimado que falta para que la emulación concluya, el tanto por ciento del progreso de la emulación y el número total de fallos detectados hasta el momento.

Con estos datos se actualizarán los campos de la interfaz gráfica para que el usuario pueda ver en tiempo real la evolución de la emulación.

Si se detecta el fin de la emulación se cerrará el canal de comunicación y se liberará el espacio reservado para las variables empleadas en esta función.

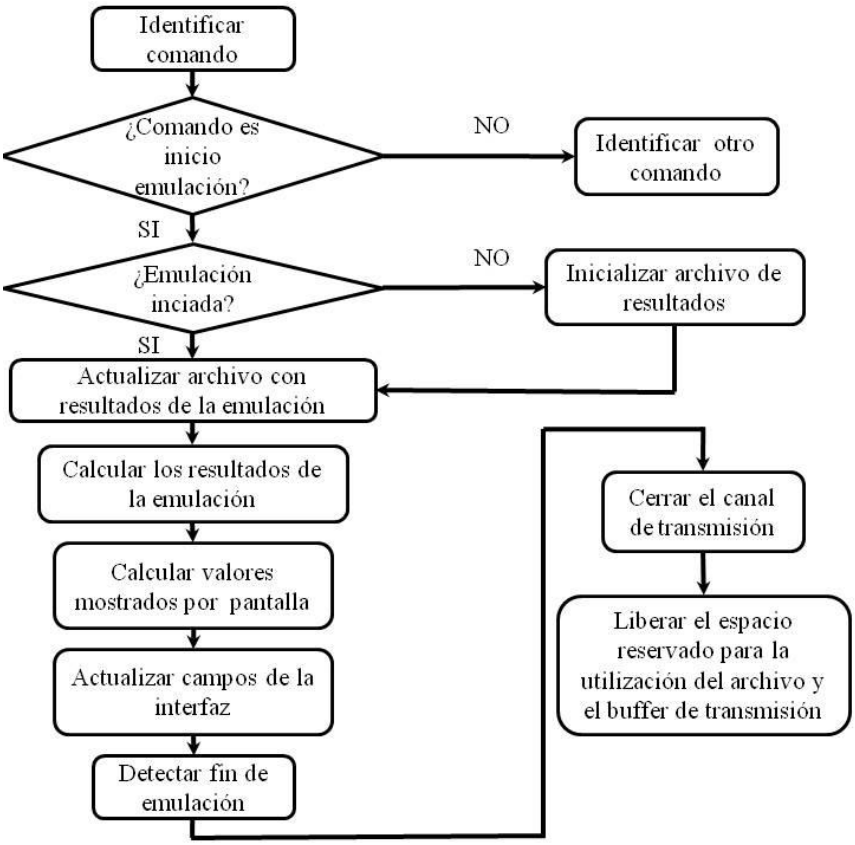

Figura 5.10: Diagrama de flujo para enviar la petición de inicio de la emulación.

Si el comando recibido no corresponde con los anteriormente expuestos se informará del error.

En el caso que se haya recibido el comando de limpiar registros, mostrar resultados ó mostrar último resultado se calcularán los parámetros necesarios para rellenar los campos de la interfaz, con los resultados de la emulación recibidos, y se actualizarán los campos de la aplicación, se cerrará el canal de comunicación y se liberará el espacio reservado para las variables empleadas en esta función.

En la Figura 5.11 se muestra el diagrama de flujo en el caso que el comando recibido sea distinto del comando de inicio de emulación.

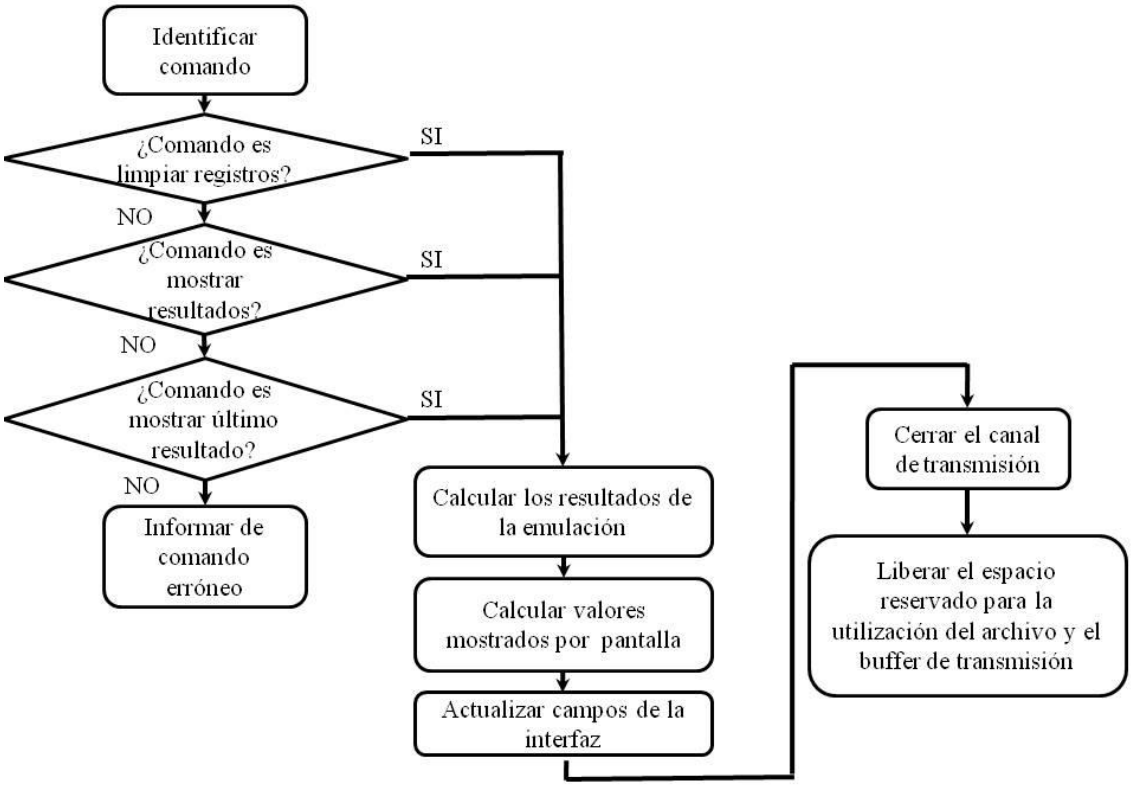

Figura 5.11: Diagrama de flujo para enviar la petición distinta a la de inicio de la emulación.

5.4. INFORMACIÓN MOSTRADA

En la pestaña de resultados se indica el progreso de la emulación. Esta pestaña está dividida en cinco secciones: resultados de la emulación, progreso de la emulación, estado de las comunicaciones USB, fallos detectados y botones de instrucciones. En la Figura 5.3 se pueden ver las distintas secciones.

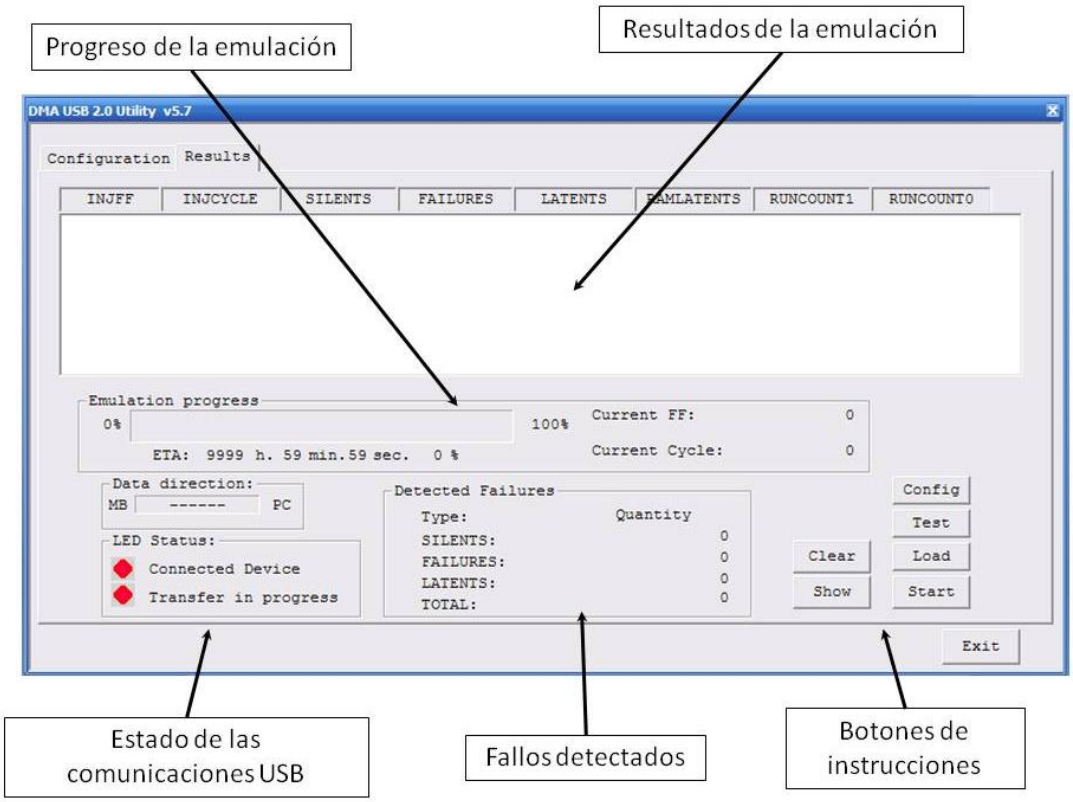

Figura 5.12: Secciones de la pestaña de resultados.

La sección de resultados de emulación es un control que se va actualizando a medida que van llegando nuevos resultados de emulación. En esta sección se van poniendo los valores de los registros que contienen información de emulación. Cada línea corresponde con los resultados del FF indicado en la primera columna, los ciclos de inyección empleados para la prueba de ese FF, los fallos "Silents", "Failures", "Latents" y "RAMLatents" detectados, además de los ciclos de reloj empleados desde el inicio de la emulación.

En la sección de progreso de la emulación se indica el tanto por ciento de FFs analizados y el tiempo estimado para concluir la actual emulación.

También se muestra en otra sección el estado de las comunicaciones USB, el sentido de los datos transmitidos, hacia el MicroBlaze ó hacia el ordenador, si hay un dispositivo conectado y si hay una transmisión en progreso.

Además se muestra en la sección de fallos detectados un resumen con el número de cada tipo de fallo detectado y el total de fallos detectados en tiempo real a medida que avanza la emulación.

Por último, la sección de botones de instrucción contiene los botones necesarios para realizar las comunicaciones con la tarjeta: "Config", "Test", "Load", "Start", "Clear" y "Show".

- Config: sirve para enviarle el archivo de configuración al microcontrolador de la tarjeta que hará de intérprete en las comunicaciones USB.
- Test: se utiliza para realizar una prueba de las comunicaciones USB.
- Load: al pulsar este botón se envían los parámetros, almacenados en el archivo correspondiente, al sistema de Emulación Autónoma.

- Start: con este botón se indica el Emulador que inicie la emulación del circuito.
- Clear: su función es indicar al MicroBlaze que realice una limpieza de los registros y envíe al ordenador el valor almacenado en los registros para comprobar que el comando se ha ejecutado correctamente.
- Show: al pulsar este botón se indica al MB que envíe los últimos resultados de emulación que haya almacenado.

5.4.1. Botón Config

Al pulsar este botón el usuario la aplicación realizará la configuración de las comunicaciones USB, para ello hará una llamada a la función ReadConfig con la se obtendrán los parámetros que se identifica la tarjeta conectada y se obtienen las direcciones de los archivos empleados por la interfaz

A continuación se fijará la dirección de transmisión para ser mostrado en la interfaz, al usuario y por último se enviará el archivo de configuración del microcontrolador que actuará de intérprete en las comunicaciones USB mantenidas entre la aplicación y el MicroBlaze. Para ello se realizará una llamada a la función ConfigureUSB.

En la Figura 5.13 se resume la funcionalidad implementada por el botón Config.

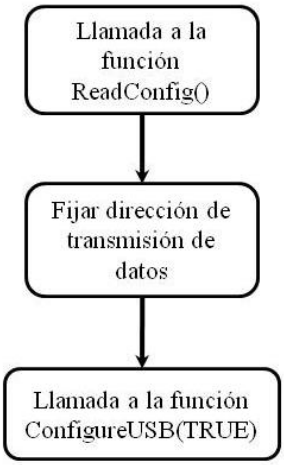

Figura 5.13: Diagrama de flujo para atención al pulsar botón Config.

5.4.2. Botón Test

Cuando el usuario desee hacer una prueba de la comunicación USB debe pulsar este botón. Como se muestra en la Figura 5.14, en primer lugar se realiza una llamada a la función TestUSBComm, la que se encarga de realizar una sencilla prueba de comunicaciones, que consiste en enviar un paquete y recibir otro, como ya se comentó previamente.

Si la función TestUSBComm devuelve un error se informará al usuario que las comunicaciones no funcionan correctamente. Y si, por el contrario, dicha función no devuelve un error se le comunica al usuario que la comunicación USB está lista para usarse.

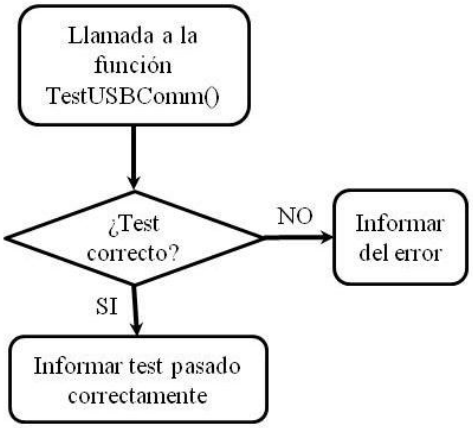

Figura 5.14: Diagrama de flujo para atención al pulsar botón Test.

5.4.3. Botón Load

Para realizar la carga de parámetros en el emulador se debe pulsar en el botón Load, a través del que se realizará todo lo necesario para enviar al MicroBlaze los parámetros de la emulación.

Como se muestra en la Figura 5.15, en primer lugar se fija la dirección de transmisión de datos para indicar al usuario que el PC está enviando datos al MB.

El envío de los parámetros se realiza a través de la llamada a la función SendTrama ya explicada anteriormente.

Una vez vuelta la ejecución a la función del botón Load se evaluará el valor devuelto por la función SendTrama, si es FALSE indicará que ha habido un error en el envío de los parámetros y se informará al usuario de dicho error. Si por el contrario, devuelve TRUE, se informará al usuario que el emulador cuenta con los parámetros de emulación y está listo para realizar la emulación al circuito.

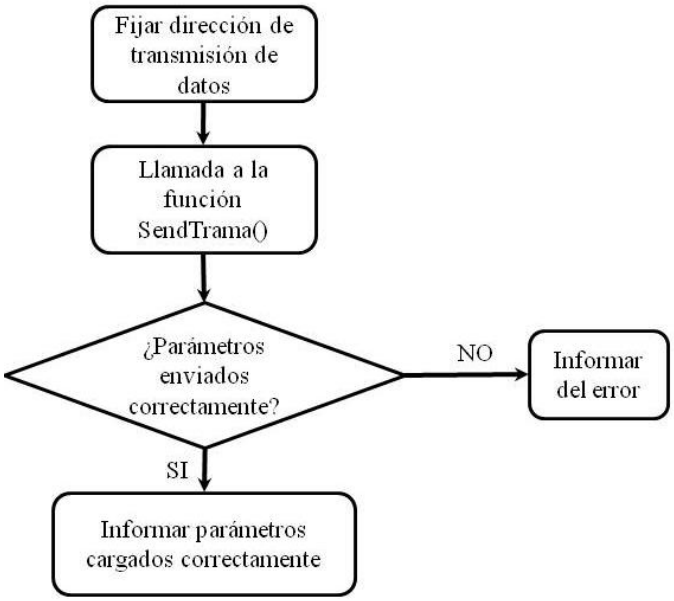

Figura 5.15: Diagrama de flujo para atención al pulsar botón Load.

5.4.4. Botón Start

Con el botón Start se inicia el proceso de emulación autónoma. Con esta función se envía el comando al MB para que este indique al emulador que comience el proceso de prueba de la robustez frente a fallos al circuito analizado. Además irá recopilando los resultados a medida que el MicroBlaze los envíe a través del puerto de comunicaciones USB habilitado para tal efecto, siendo estos datos tratados por la aplicación. Para la implementación de la funcionalidad se ha optado por un diseño sencillo de esta función añadiendo más complejidad a la función que es llamada desde esta, aprovechando el diseño modular con el que se ha desarrollado toda la interfaz gráfica de usuario.

A continuación, en la Tabla 5.9, se exponen las variables empleadas por esta función que implementa lo referente al inicio de la emulación y la adquisición de los resultados.

Esta función comienza con la declaración y la inicialización de las variables que utiliza. Posteriormente se actualizarán los campos de la interfaz gráfica, y tras lo que, abrirá el archivo de resultados en modo escritura, de manera que si hubiese datos anteriores almacenados en es fichero, serían sustituidos por los que se le van a escribir a lo largo de la emulación.

Para tener una referencia temporal se almacena la hora actual en la que ha comenzado la emulación, siendo éste un dato indicado en el archivo de resultados. Como se quiere comprobar que los registros del emulador están vacíos se le enviará el comando de mostrar resultados almacenados en la constante SHOW_RESULTS. Y se fija la dirección de transmisión de datos del PC al MB. Una vez realizado todo lo anterior se llama a la función ReadTrama para que esta se encargue, tanto de enviar el comando al MicroBlaze, como de almacenar los valores guardados en los registros del emulador.

Como ya va comenzar el proceso de emulación autónoma se deshabilitan todos los botones para que no haya conflictos en el funcionamiento de la interfaz.

Se vuelve a enviar otro comando, pero esta vez se fija el comando de comienzo de la emulación, almacenado en la constante START_EMULATION. La aplicación entrará en un bucle del que sólo saldrá en el caso que la emulación haya concluido. Dentro de este bucle se realizará una llama a la función ReadTrama la cuál irá obteniendo los resultados de la emulación y los irá procesando.

En el momento que se detecte el fin de la emulación, faltará un último resultado, por lo que se le enviará el comando de petición de resultados analizando que el resultado recibido sea el último y en ese momento se dará por concluida la emulación.

Para terminar se vuelve a habilitar la funcionalidad de todos los botones para que el usuario pueda interactuar con la aplicación y se cerrará el archivo de resultados para que pueda ser leído.

En la Figura 5.16 se muestra un diagrama de flujo donde se resume la funcionalidad de la función del botón Start.

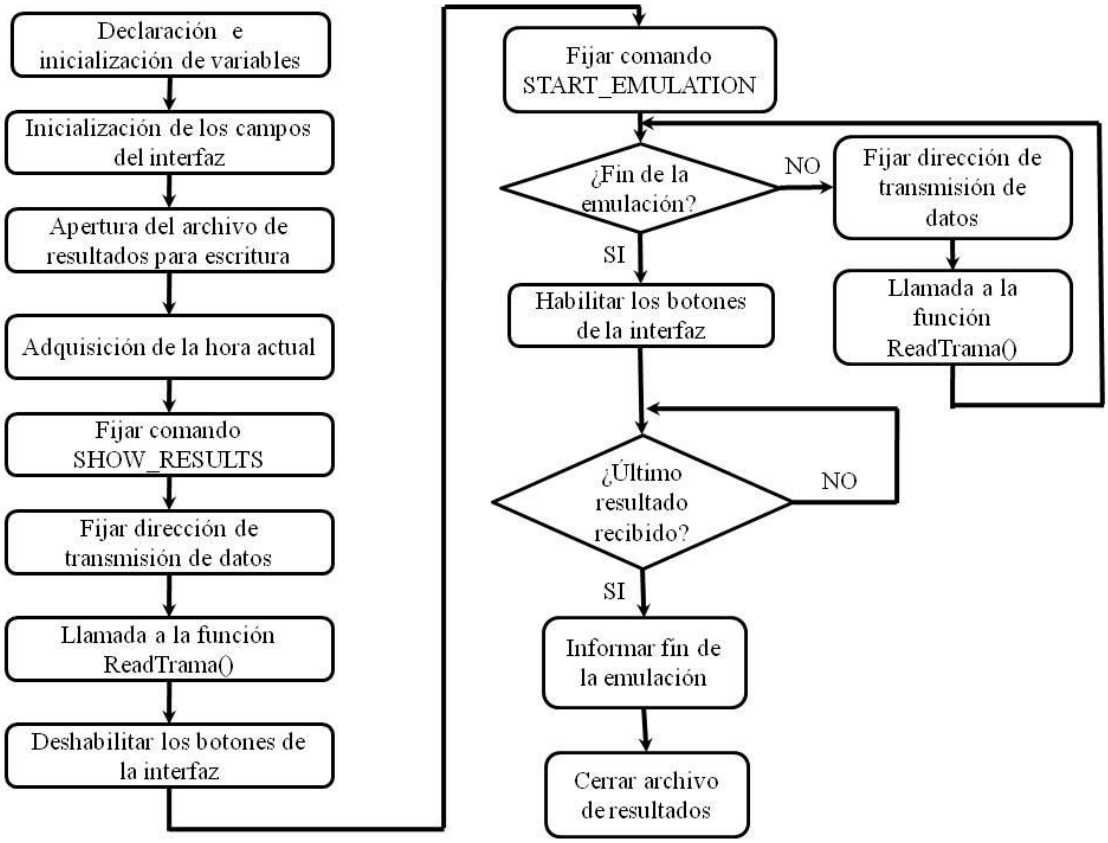

Figura 5.16: Diagrama de flujo para atención al pulsar botón Start.

5.4.5. Botón Clear

La funcionalidad del botón clear esta ideada para enviar la orden al MicroBlaze, para que éste se encargue de limpiar los registros del emulador. Por lo que esta función es capaz de enviar un comando a través del puerto USB. Además realizará una petición de resultados para comprobar que los registros no tienen valores distintos de cero, lo que será mostrado a través de la interfaz.

Primero se fija el comando de limpiar registros almacenados en la constante CLEAR_REGISTERS, para después hacer una llamada a la función ReadTrama que es la que se encarga de las comunicaciones bilaterales con el MB.

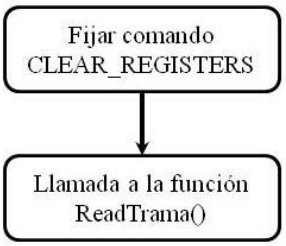

Figura 5.17: Diagrama de flujo para atención al pulsar botón Clear.

5.4.6. Botón Show

La función implementada por el botón Show es similar a la desarrollada en la función del botón Clear, con la salvedad que al pulsar este botón sólo se realizará una lectura de los registros del emulador.

La diferencia fundamental es el comando empleado, que es el comando de petición de resultados, almacenado en la constante SHOW_RESULTS, y como en el caso anterior, tras fijar el comando se realiza una llamada a la función ReadTrama, la que se encargará de las comunicaciones y que los resultados pedidos por el usuario se muestren en la interfaz gráfica.

Figura 5.18: Diagrama de flujo para atención al pulsar botón Show.

5.5. TRATAMIENTO DE RESULTADOS

Como ya se ha comentado a lo largo de este capítulo el fin de la interfaz gráfica de usuario es, por un lado, establecer comunicaciones USB entre el ordenador y el MicroBlaze, el que a su vez se comunicará con el emulador para poder hacer llegar los resultados de la emulación al PC. Además se desarrolla un tratamiento de los resultados obtenidos, para realizar un resumen en tiempo real mostrado por la aplicación, y por último, obtener un fichero el cuál contendrá los resultados de la emulación.

Todo lo referente a las comunicaciones USB se ha tratado a lo largo de este capítulo, explicando las distintas funciones que han sido implementadas para cumplir con ese objetivo, además se ha comentado también qué función se encarga de recopilar los resultados de la emulación y realizar unos cálculos para poder mostrar por pantalla, en la aplicación, tanto un resumen de los fallos detectados por el emulador, como el tanto por ciento de emulación realizada en tiempo real, como el tiempo estimado para alcanzar el fin de la emulación, como el proceso mediante el que se almacenan los resultados de la emulación en un fichero.

En este capítulo se pretende explicar de forma más detallada los cálculos realizados para el cálculo del tiempo estimado de fin de emulación, y el archivo de resultados.

5.5.1. ESTIMACIÓN DEL TIEMPO DE FIN DE EMULACIÓN

El cálculo del tiempo estimado en finalizar la emulación se basa en una interpolación lineal, a partir de los datos de tiempo obtenidos, a medida que va transcurriendo la emulación. En primer lugar, se obtiene el momento en el que se incia la emulación el que será usado como referencia.

A medida que vayan llegando resultados se calculará el incremente de tiempo desde el momento inicial al momento en el que se obtiene un nuevo resultado. Para calcular los incrementos se vierten tanto horas, como minutos a segundos. Denominando x al tiempo que tarda en obtener

resultados de n Flip Flops y siendo N el número total de Flip Flops para la campaña actual, se ha utilizado la fórmula mostrada a continuación:

$$\text{Tiempo restante} = x \cdot N / n$$

La sencillez de este método aporta rapidez de cálculo, pero a la vez, con los primeros resultados obtenidos de las emulaciones un error en la estimación grande, error que a medida que avance la emulación será minimizado, dando valores más fiables a medida que la emulación avanza en su campaña.

5.5.2. ARCHIVO DE RESULTADOS

El archivo de resultados consta de cuatro partes: hora de inicio de la emulación, parámetros de la emulación, resultados de la emulación y, por último, hora de fin de la emulación.

Es interesante, en un archivo donde se exponen los resultados de la emulación tener una referencia temporal de cuánto duró, para poder hacer predicciones en campañas futuras y, de esta manera, dar fecha de entrega para proyectos cuyo fin sea entregar unos resultados de emulación de circuitos. Por ello se incluye en la cabecera la hora de comienzo de la emulación y en el final del archivo la hora de fin de la campaña realizada.

Como comprobación que los parámetros cargados en el emulador son los correctos, y que dicho sistema de emulación autónoma inicia la emulación con los registros a cero, se ha incluido una zona del archivo con los parámetros de emulación y con los valores de los registros donde se almacenan los resultados. Además, si este archivo es analizado en un futuro, los datos que contiene el fichero pueden ser suficientes para identificar el tipo de campaña que se llevó a cabo.

Al final del resumen se muestran los resultados de la emulación. En la Figura 5.19 se muestra un ejemplo de archivo de resultados. Se han omitido los resultados entre el Flip Flop 0x00000004 hasta el Flip Flop 0x0000005B, números en hexadecimal por simplicidad. Los resultados constan de el registro de control, el registro de estado, el registro que contiene el número Flip Flop al que corresponden esos resultados, el número de ciclos de inyección que han sido utilizados para la prueba frente a fallos de este biestable, los fallos detectados, silenciosos, fallos, latentes, latentes en RAM, y por último utilizando dos registros de 32 bits cada uno el número de ciclos de reloj empleados en realizar las pruebas desde el inicio de la emulación, cuyo valor es cero hasta el biestable correspondiente.

Un dato que llama la atención al echar un vistazo al archivo de resultados es que se observa como FF de inicio el 0 y FF de fin el 100 (0x64 en hexadecimal) en el resumen de los parámetros. Sin embargo, en los resultados se ve que la numeración de los biestables comienza en 1 y finaliza en 101 (0x65 en hexadecimal). Esto se debe a que para el emulador en los parámetros se le debe indicar un rango de que comprende desde 0 hasta n, y los resultados que devuelve pertenecen a un rango desde 1 hasta n+1.

Hora de comienzo de la emulacion: 19:59
Parametros de la emulacion:

RCONTROL	RSTATUS	FFBEGIN	FFEND	CYCLEBEG	CYCLEEND	TBCYCLES
00000001	00000000	00000000	00000064	00000000	00002259	0000225A

RATE	LENGTH	OFFSET	ITERATIONS
00000003	00000008	00000004	00000003

INJ_FF	INJ_CYCLE	Silent	Failure	Latent	RAMLat	ClkCount1	ClkCount0
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000

Resultados de la emulacion:

RCONTROL	RSTATUS	INJ_FF	INJ_CYCLE	Silent	Failure	Latent	RAMLat	ClkCount1	ClkCount0
00000001	00000011	00000001	00002259	0000164D	00000AF0	0000006D	000000B0	00000000	0000B167
00000001	00000011	00000002	00002259	00002C9A	000015DF	000000DA	00000161	00000000	00001157
00000001	00000011	00000003	00002259	000042E3	000020D1	00000147	00000213	00000000	00003D15
00000001	00000011	0000005C	00002259	0000E3EE	00000332	00002F38	00004200	00000001	00009C71
00000001	00000011	0000005D	00002259	0000FC8B	00000A69	00003065	00004359	00000001	00003671
00000001	00000011	0000005E	00002259	000014AC	0000123D	00003191	00004492	00000001	0000CB7D
00000001	00000011	0000005F	00002259	00002D76	000019B4	000032C8	00004574	00000001	0000E1E9
00000001	00000011	00000060	00002259	00004A5F	00001A84	000033F5	000048E8	00000001	000048D7
00000001	00000011	00000061	00002259	000066D6	00001B5C	00003525	00004CC3	00000002	000080B7
00000001	00000011	00000062	00002259	00008813	00001B5C	00003642	00004CC3	00000002	00007E93
00000001	00000011	00000063	00002259	0000A44C	00001F02	00003763	00004E1D	00000002	0000D80F
00000001	00000011	00000064	00002259	0000BCA4	00002725	0000388F	00004ED0	00000002	0000CAA1
00000001	00000011	00000065	00002259	0000D4FC	00002F48	000039BB	00004F83	00000002	0000BD33

Hora de fin de la emulacion: 20:03

Figura 5.19: Archivo de resultados.

INTEGRACIÓN DEL SISTEMA COMPLETO

6. INTEGRACIÓN DEL SISTEMA COMPLETO

Este capítulo resume la integración del sistema completo, presentando un ejemplo de aplicación de una emulación, desde la conexión de la tarjeta al ordenador, equipo se va a utilizar para almacenar el archivo de resultados, pasando por la carga del archivo .bit (bitstream) a través de la aplicación Xilinx Platform Studio, finalizando con una guía paso a paso del uso de la aplicación de interfaz de usuario (GUI).

6.1. CONEXIÓN DE LA TARJETA CON EL PC

La tarjeta presenta dos conexiones con el ordenador. En primer lugar, un conexión mediante la que se implementan las comunicaciones serie, con protocolo RS-232, que se conecta con el PC a través de su puerto de comunicaciones serie. Esta conexión es empleada para la depuración en caso de un mal funcionamiento del dispositivo y además, fue usado a lo largo de la realización del proyecto para ese mismo fin, la depuración. Por otro lado, se conecta el puerto USB del ordenador con la tarjeta para realizar las comunicaciones USB que se llevan a cabo a lo largo de la campaña de emulación de circuitos.

En la Figura 6.1 se puede ver la tarjeta Virtex 4 Evaluation Board conectada con el PC del laboratorio de microelectrónica, donde se ha llevado a cabo el desarrollo de este Proyecto Fin de Carrera. En la misma figura se identifica cada cable de comunicaciones y los dispositivos intervinientes en este proyecto.

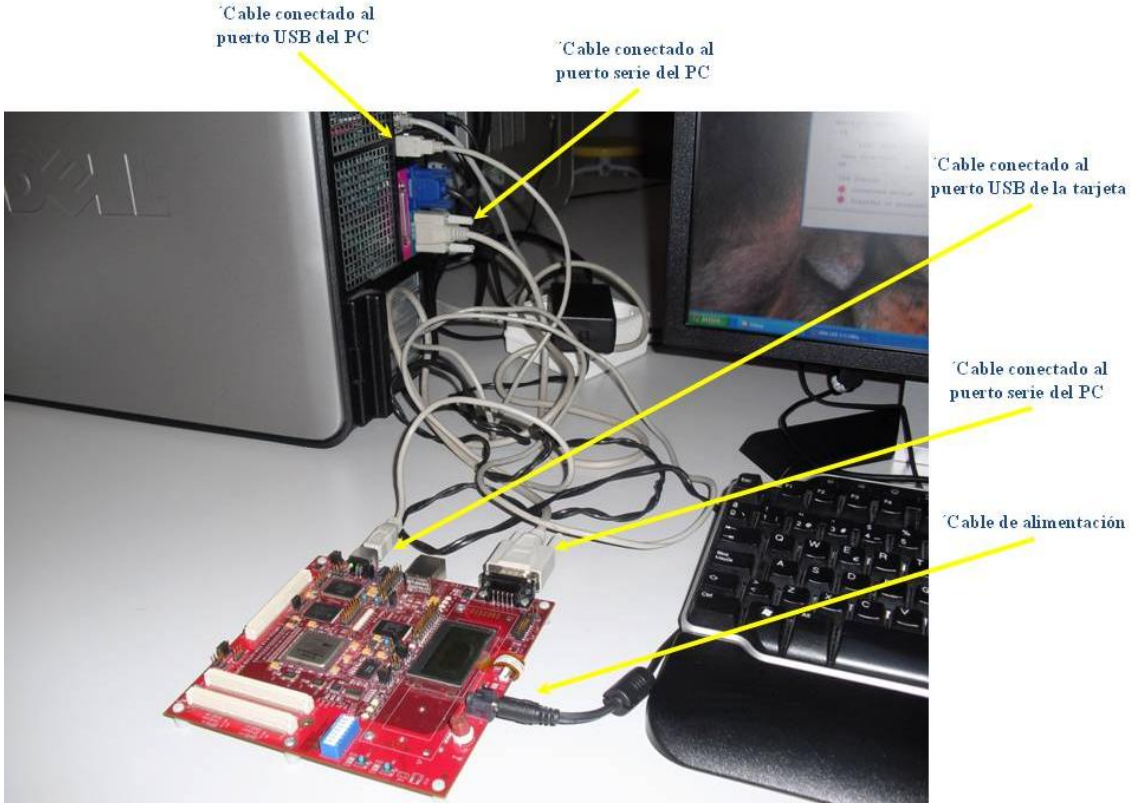

Figura 6.1: Conexiones de la tarjeta con el PC.

6.2. CARGA DEL BITSTREAM EN LA TARJETA

Como ya se ha comentado anteriormente se ha utilizado la aplicación de Xilinx XST para realizar la integración del sistema de emulación autónoma en un entorno que contiene un microcontrolador MicroBlaze encargado de manejar las comunicaciones entre ordenador y emulador, para por un lado facilitarle los parámetros de emulación al emulador, y por otro lado, pasarle los resultados de la emulación al PC, siendo el protocolo de comunicaciones utilizado USB 2.0.

Una vez realizado el proyecto de integración del emulador en el entorno del MB e incluidos todos los archivos de código necesarios para la ejecución del MicroBlaze, pulsando el botón de carga del bitstream a la FPGA, la aplicación XST realizará los pasos necesarios para generar y posteriormente cargar el bitstream en la FPGA.

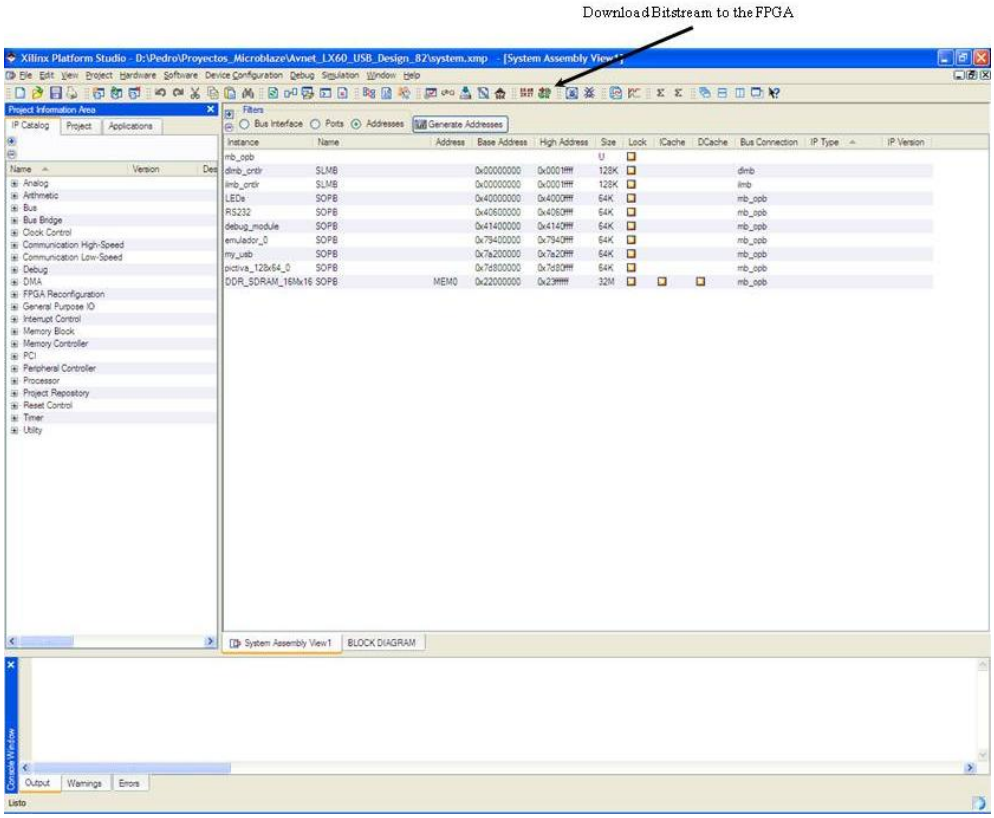

Figura 6.2: Carga del bitstream con el Xilinx Platform Studio.

Una vez cargado correctamente el bitstream, el MicroBlaze enviará a través del puerto serie un mensaje indicando que está configurado correctamente. Para el desarrollo de este proyecto, como ya se ha contado en capítulos anteriores, se ha empleado el programa Tera Term Web 3.1. En la Figura 6.2 se muestra el mensaje reflejado en programa que controla las comunicaciones con protocolo RS-232 del PC.

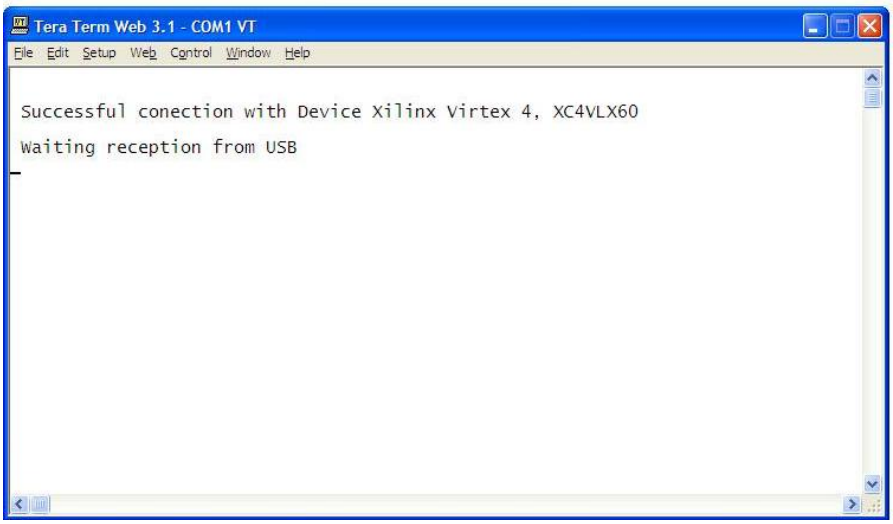

Figura 6.3: Mensaje recibido del MicroBlaze a través del puerto serie.

Tras comprobar que la FPGA contiene todos los componentes necesarios del sistema de Emulación Autónoma, y que el MB ejecuta el programa correctamente, la tarjeta está lista para realizar la emulación del circuito en cuestión.

En la figura 6.4 se ve un ejemplo de conexión de la tarjeta con el PC para proceder a cargar el bitstream en la FPGA.

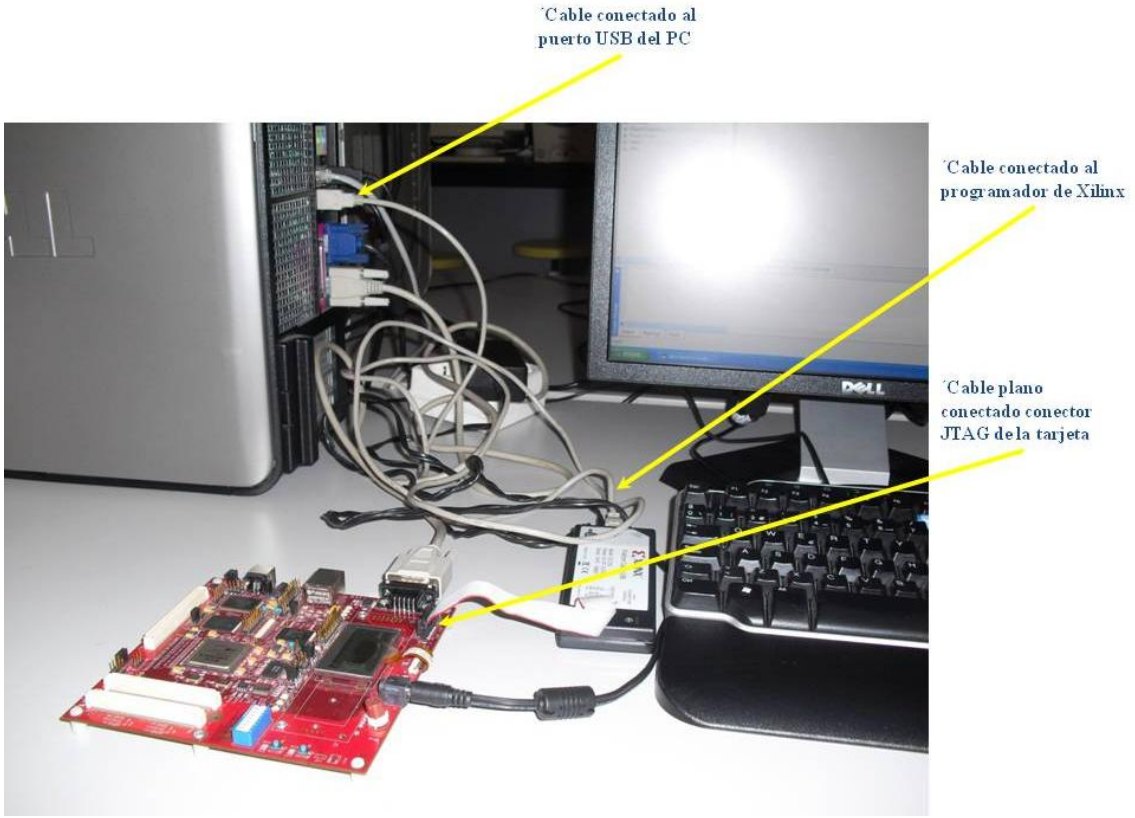

Figura 6.4: Conexiones entre el PC y la tarjeta para cargar el bitstream.

6.3. PROCEDIMIENTO DE EMULACIÓN CON INTERFAZ

La interfaz de usuario cuenta con dos pestañas. Una pestaña de configuración de la interfaz, mostrada en la Figura 6.4, y otra pestaña de resultados mostrada en la Figura 6.5.

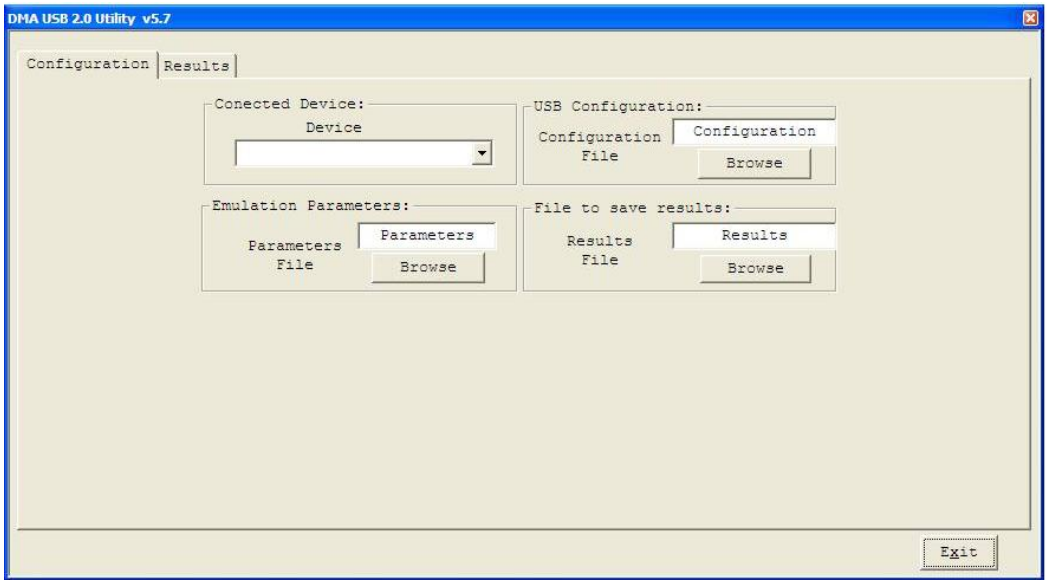

Figura 6.5: Pestaña de configuración de la interfaz.

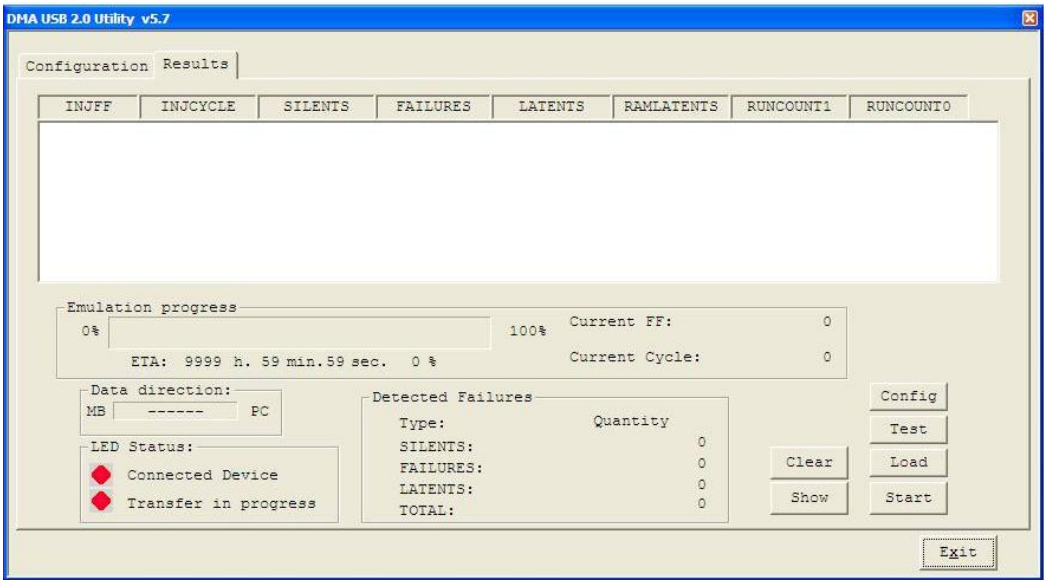

Figura 6.6: Pestaña de resultados de la interfaz.

El primer paso para utilizar el interfaz es realizar su configuración. Para ello se le debe indicar a la aplicación dónde encontrar los archivos que va a utilizar y la tarjeta que se le va a conectar. Todo esto se realiza en la pestaña de configuración. Pinchando en el botón Browse correspondiente para cada fichero se deberá seleccionar la ubicación de los mismos y por último, mediante el desplegable de la zona superior izquierda se seleccionará el tipo de tarjeta empleada. Si no se le indica la ruta de algún archivo ó la tarjeta conectada, la aplicación utilizará unos valores predeterminados.

En el ejemplo planteado en este capítulo se seleccionará la ruta de los archivos a utilizar y se seleccionará la tarjeta con la que se ha desarrollado este proyecto, la tarjeta Virtex 4 Evaluation Board. Se muestra un resumen de la configuración en la Figura 6.6.

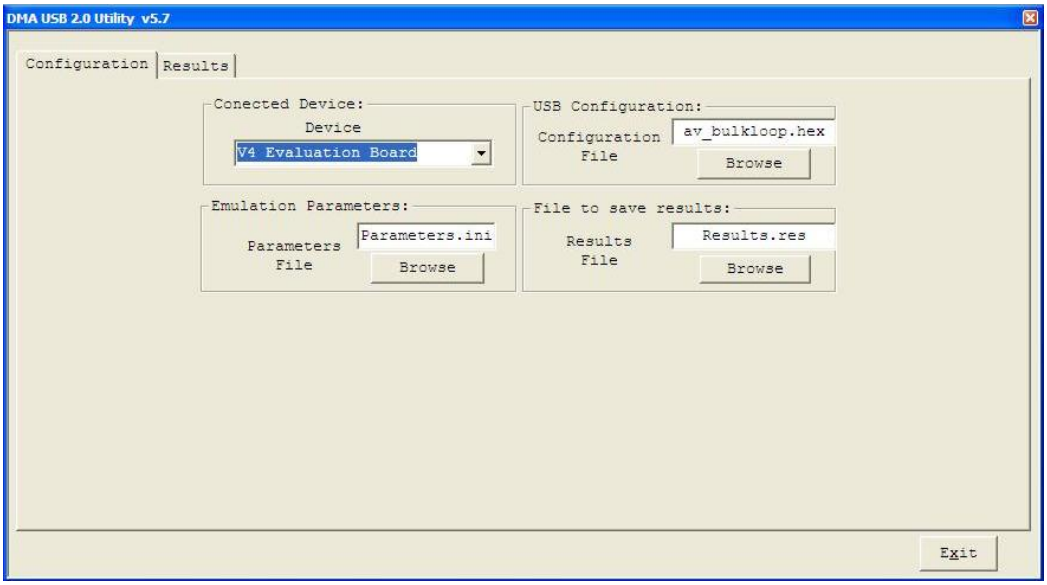

Figura 6.7: Parámetros de configuración de la aplicación.

Los siguiente pasos se realizarán en la pestaña de resultados, que se desarrollo la emulación de los circuitos y la toma de los resultados de emulación. En primer lugar se pulsará el botón Configuration, botón cuyo cometido es configurar el microcontrolador para que el MicroBlaze pueda comunicarse a través de USB con el ordenador. Pinchando en este botón la aplicación le enviará el archivo av_bulkloop.hex al microcontrolador, archivo desarrollado por la empresa Avnet, con el que se configura de manera que esté enviando y recibiendo paquetes en modo Bulk continuamente.

Si el microcontrolador ha recibido correctamente el archivo de configuración la aplicación informará con el mensaje mostrado en la Figura 6.6, en caso contrario informará con un mensaje de error.

Figura 6.8: Pestaña de resultados de la interfaz.

A continuación se debe comprobar el correcto funcionamiento de las comunicaciones USB, para ello se deberá pulsar en el botón Test. En este momento comenzará el Test de las comunicaciones USB. Si falla, la aplicación mostrará el correspondiente mensaje de error, si por el contrario el test es pasado correctamente, mostrará el mensaje mostrado en la Figura 6.7.

Figura 6.9: Test de comunicaciones USB pasado correctamente.

Una vez comprobada la comunicación USB se procede al envío de los parámetros, para que el MicroBlaze los cargue en los registros del emulador. Esto se realiza mediante el botón Load, que la pulsar el usuario sobre él, se le enviarán a través de USB los parámetros de emulación al MB. Si la aplicación ha encontrado el fichero de parámetros informará al usuario que va a comenzar con el envío de los parámetros, mensaje mostrado en la parte superior de la Figura 6.8 y si los parámetros son recibidos correctamente también será informando el usuario con el mensaje mostrado en la parte inferior de la misma figura.

Figura 6.10: Mensajes durante la carga de parámetros de emulación.

Con la comprobación del buen funcionamiento de las comunicaciones, y los parámetros de emulación cargados correctamente ya es posible comenzar la emulación, pero es aconsejable cerciorarse que los registros de resultados del emulador no contienen valores distintos de cero, para ello, se puede pulsar sobre el botón Clear, mediante el que se limpiar dichos registros y se mostrarán los valores almacenados. También se podría pulsar sobre el botón Show con el que la aplicación muestra en la pestaña de resultados los valores almacenados en los registros de resultados del sistema de Emulación Autónoma, pero si los registros tuviesen valores no nulos habría que utilizar la funcionalidad del botón Clear.

Para iniciar la emulación se ha de pulsar sobre el botón de Start, y en ese momento el sistema de Emulación Autónoma comenzará con la emulación del circuito en cuestión. En primer lugar mostrará los valores almacenados en los registros de resultados, como se ve en la Figura 6.9.

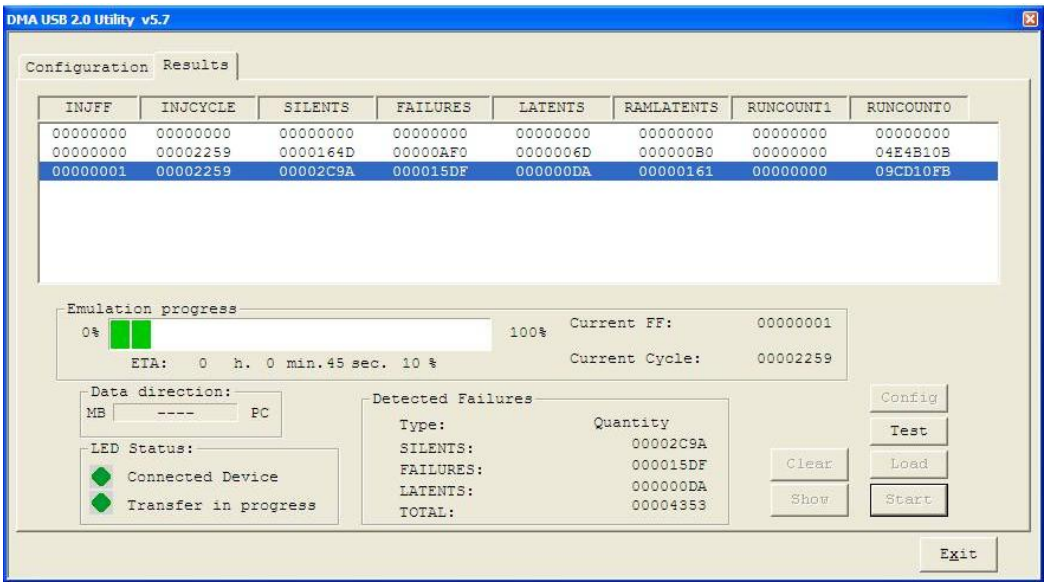

Figura 6.11: Inicio de la emulación.

A medida que avanza la emulación se irán mostrando los resultados de la emulación en las distintas zonas de la aplicación. Los valores almacenados en los registros de resultados, el tanto por ciento de emulación realizada, el tiempo estimado para el fin de la emulación, el FF al que

corresponden los últimos resultados mostrados, en número de ciclos empleados en el análisis de ese Flip Flop, y un resumen de los tipos de fallos detectados, además del número total de fallos detectados hasta el momento. Como información adicional se indica la dirección de los datos y el estado de las comunicaciones, si hay un dispositivo conectado y si se está llevando a cabo una transmisión.

Cuando la emulación concluye la aplicación tiene un aspecto similar al mostrado en la Figura 6.10, qué es resultado de la emulación de ejemplo empleada para el desarrollo de este capítulo. Volviendo todos los botones a tener funcionalidad.

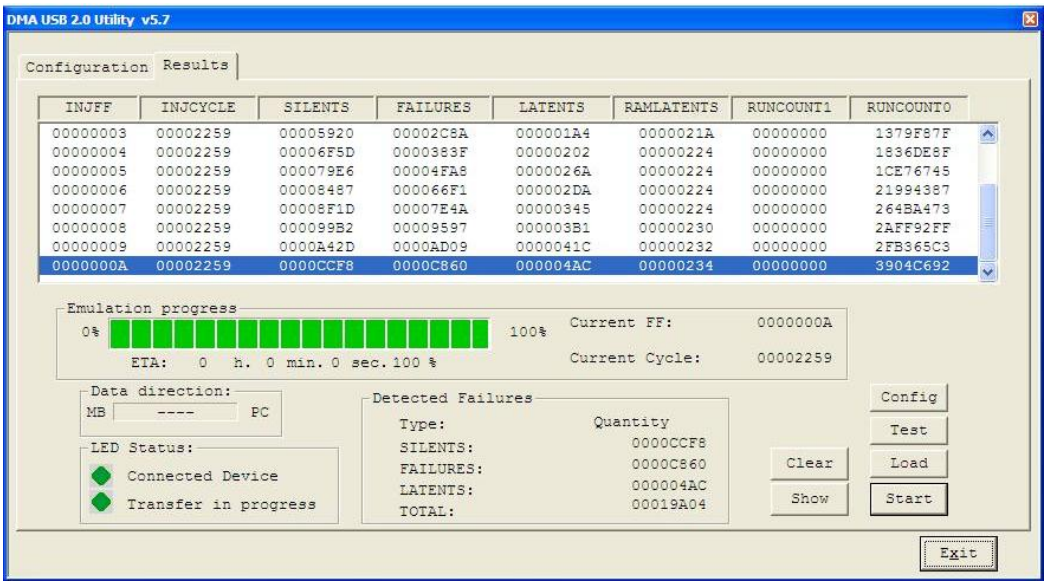

Figura 6.12: Fin de la emulación.

Como ejemplo de uso se han incluido dos figuras más, la Figura 6.11 y la Figura 6.12. La primera es el resultado de pulsar el botón show, como se ve en la misma se muestran los últimos resultados almacenados en los registros de resultados del emulador, los que coinciden con los resultados obtenidos tras analizar el último FF de la campaña llevada a cabo.

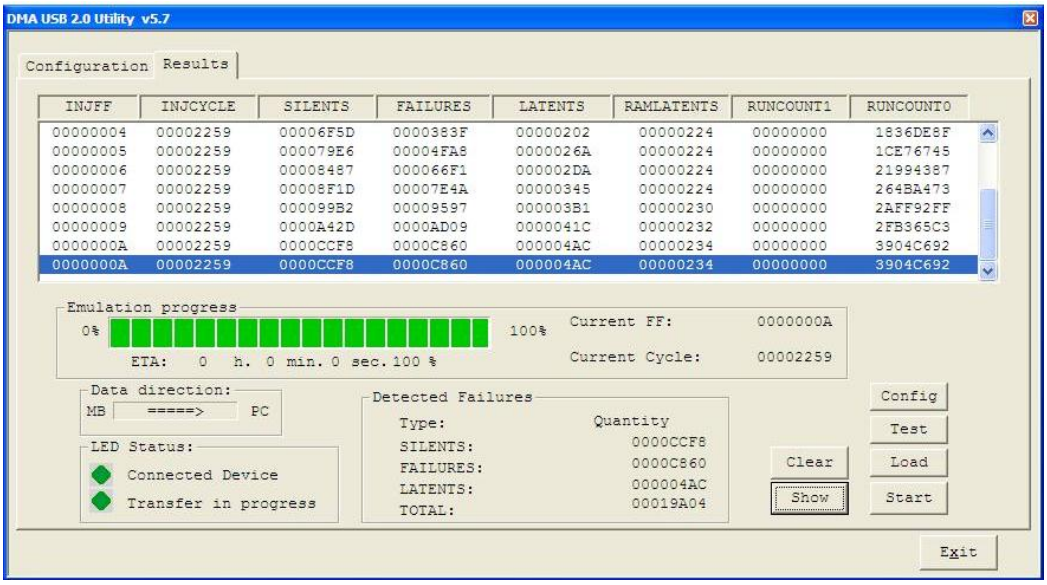

Figura 6.13: Resultado de pulsar el botón Show.

La segunda figura muestra el resultado de pulsar el botón Clear, primero se limpian los registros del emulador y para comprobar que se ha realizado correctamente, se muestran los valores guardados en los registros de resultados del emulador.

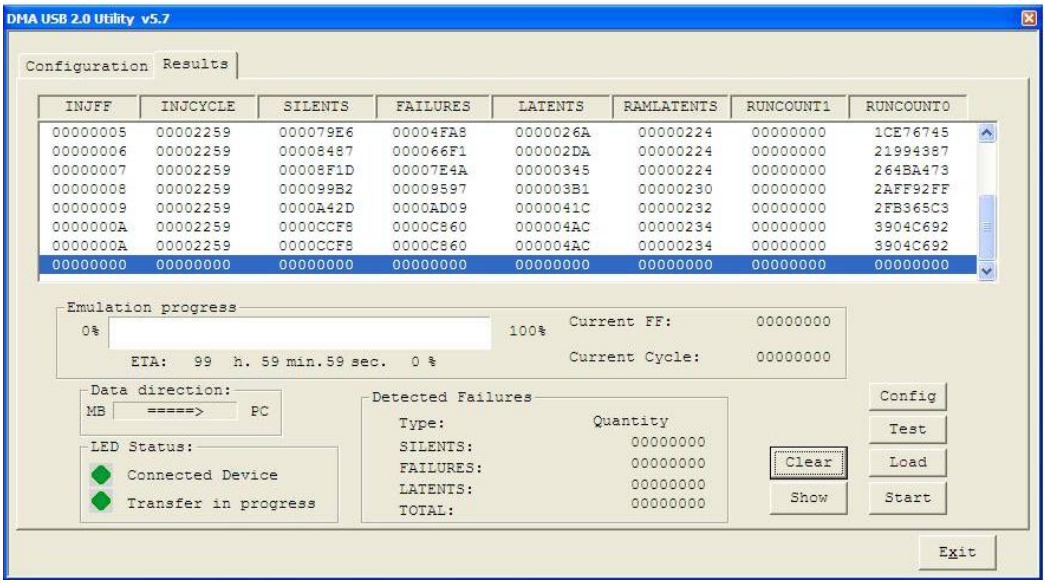

Figura 6.14: Resultado de pulsar el botón Clear.

Al finalizar una emulación se obtendrá por un lado el resumen mostrado por la aplicación. Y por otro lado, un archivo en el que se han ido almacenando los resultados a medida que se ha ido ejecutando la emulación, el fichero Results.res.

PRESUPUESTO

7. PRESUPUESTO:

A continuación se exponen costes imputados a la realización del Proyecto Fin de Carrera. Los costes se han clasificado en dos tipos: por un lado los costes debidos al material empleado, y por otro lado, los costes debidos a la mano de obra de personal especializado.

7.1. MATERIAL:

Se ha estimado un coste de 1.000€ para el PC empleado a lo largo de la vida del proyecto. También se ha incluido el coste de la tarjeta que contiene la FPGA y los periféricos empleados en el desarrollo del proyecto, tarjeta de desarrollo Virtex 4 LX60 Evaluation Board por un valor de 1.500€. Y por último el coste de las herramientas, el Software de la empresa Xilinx, ISE Suite Embedded Edition, con un coste de 3.000€, y el Software para desarrollo de aplicaciones de interfaz gráfica, Visual Studio 2008, con un coste de 550€.

Concepto	**Coste(€/ud)**	**Unidades**	**Coste(€)**
PC	1.000	1	1.000
Virtex 4 LX Evaluation Board	1.500	1	1.500
Software de Xilinx, ISE Suite, Embedded edition.	3.000	1	3.000
Visual Studio 2008	550	1	550
TOTAL			**6.050**

7.2. PERSONAL:

En lo referente al personal empleado, es necesario contar con un Ingeniero Superior para encargarse de la fase de diseño de la arquitectura Hardware del sistema, así como el diseño de la arquitectura software, que repercute con un coste de 21.000€/año de salario. Como es necesaria una fuerza de trabajo equivalente a 30 semanas, se ha estimado dividiendo el salario entre 52 semanas que tiene un año lo que resulta, un salario de 403,85€/semana, imputando un coste total al proyecto de 12.115,38€.

Además es necesario el contrato de una persona con conocimientos de programación. Esta persona encargada de programar la interfaz gráfica de usuario se le paga un salario equivalente de 18.000€/año, contando con 52 semanas al año, implica un coste de 346,5€/semana. Como son necesarias 24 semanas para desarrollar su trabajo, se incurre en un coste de 8.307,69€.

Concepto	**Coste(€/semana)**	**Semanas**	**Coste(€)**
Ingeniero Superior	403,85	30	12.115,38
Programador	346,5	24	8.307,69
TOTAL			**20.423,07**

7.3.COSTE TOTAL PARA LA REALIZACIÓN DEL PROYECTO:

Al sumar las cantidades correspondientes a los costes de material y de personal, se obtiene el coste total del proyecto sin I.V.A., a este conste se le incluye el Impuesto sobre el Valor Añadido que toma el valor del 18% del importe, que hace un coste total de 31.238,22€.

Concepto	**Coste(€)**
Material	6.050
Personal	20.423,07
COSTE TOTAL PROYECTO (sin I.V.A.)	26.473,07
I.V.A.(18%)	4.765,15
COSTE TOTAL PROYECTO	**31.238,22**

CONCLUSIONES Y TRABAJOS FUTUROS

8. CONCLUSIONES Y TRABAJOS FUTUROS

8.1. VISIÓN GLOBAL DEL TRABAJO REALIZADO

Llegados a este punto se puede realizar una valoración global del Proyecto Fin de Carrera. La realización de este proyecto ha abarcado una amplia gama de aspectos:

- Desarrollo de un sistema basado en el microcontrolador de Xilinx, MicroBlaze.
- Descripción de hardware mediante el lenguaje VDHL para la integración del emulador en el entorno del MB.
- Desarrollo del software que ejecuta el microcontrolador.
- Implementación de una interfaz gráfica de usuario a través del programa Visual Studio 2008.

Al fin del proyecto se presenta un sistema completo de emulación autónoma que transmite los resultados mediante el protocolo de comunicaciones serie USB. Además está formado por un microcontrolador empotrado en la FPGA.

8.2. ASPECTOS FAVORABLES

Como aspectos favorables, y tras los resultados experimentales obtenidos, se puede afirmar que las comunicaciones ya no son un cuello de botella en el sistema. Se han realizado varias campañas de emulación para corroborarlo, campañas que al emplear comunicación serie con protocolo RS-232, si presentaba la problemática de perder resultados de la emulación. Como valor cuantitativo se ha obtenido una tasa de transmisión de 20 veces más resultados obtenidos en el mismo tiempo del nuevo sistema frente al antiguo, en una campaña en la que con el sistema antiguo se perdían resultados.

Otro aspecto a destacar es la GUI desarrollado que da un aspecto más amigable para el usuario, a la vez de ser más intuitivo el proceso de emulación una vez cargado en la FPGA el sistema de Emulación Autónoma. La GUI aporta también un resumen de los resultados, en tiempo real, dando al usuario una idea, de lo fiable que es el circuito y del tiempo que puede faltar para que la emulación concluya.

8.3. ASPECTOS DESFAVORABLES

Como aspectos desfavorables, este proyecto presenta los siguientes inconvenientes:

- Es un sistema poco flexible, ya que sólo es válido para el Sistema de Emulación Autónoma que se tomó como referencia, y a partir del que se evolucionó hasta esta versión. Siendo complicado sin modificar el diseño adecuarlo a otro sistema de Emulación Autónomo que presente unas características distintas. Sobre todo si son necesarios más registros para los parámetros, ó si los resultados de emulación cambiasen de nombre ó fuesen necesarios más registros para contar con un mayor número de resultados.
- Otro problema que es probable que aparezca es el que se achacaba al emulador predecesor, que las comunicaciones vuelvan a ser un cuello de botella.

8.4. TRABAJOS FUTUROS:

Estudiando los aspectos desfavorables detectados se plantean dos líneas de trabajos futuros:

- Rediseñar el Sistema completo para hacerlo más flexible, de modo que la integración de un nuevo emulador no implique ninguna modificación, ni por parte del sistema formado alrededor del MicroBlaze, ni por parte de la Interfaz Gráfica de Usuario.

- Incrementar la velocidad de transmisión empleando un canal de comunicaciones Ethernet, lo que daría un margen mucho mayor que el actual, e incluso mayor al sucesor del USB 2.0, el USB 3.0.

ANEXOS

ANEXO I: CÓDIGO DE LAS FUNCIONES DE MICROBLAZE

Bibliotecas y variables globales

```
#include <EmulacionApp.h>
#include "xparameters.h"
#include "types.h"
#include "microblaze_interrupts_i.h"
#include "xbasic_types.h"
#include "xstatus.h"
#include "xio.h"

// Global variables defined by user
Xuint32 EmuladorExpired = 0;
Xuint16 NuevoFF = 0x0000;
```

Función main

```
int main(void)
{

  Xuint8 i;
  Xuint8 FinEmulacion;
  Xuint8 endp_num = 0;
  Xuint16 flags;
  Xuint16 data_received;
  Xuint16 *instrucciones;
  Xuint16 *registros;
  Xuint32 LastEmuladorExpired = 0x00000000;
  Xuint8 Peticiones = 0x00;
  Xuint32 INJFF = 0x00000000;
  Xuint32 FFEND = 0x00000000;

  instrucciones = calloc(6,sizeof(Xuint16));
  registros = calloc(48,sizeof(Xuint16));

  for (i = 0; i < 6; i++)
   instrucciones[i] = 0x0000;

  FinEmulacion = 0x00;

   // Inicialization of EMULATOR
   EMULADOR_mReset((void *)XPAR_EMULADOR_0_BASEADDR);

   xil_printf("\033[H\033[J"); //clears the screen

   // Mensaje por pantalla para comprobar la comunicacion serie
   xil_printf("\r\n\r\n Successful conection with Device Xilinx
Virtex 4, XC4VLX60 \r\n");
```

```
   // Inicializacion de la Cache del MicroBlaze
   #if XPAR_MICROBLAZE_0_USE_ICACHE
    microblaze_init_icache_range(0,
XPAR_MICROBLAZE_0_CACHE_BYTE_SIZE);
    microblaze_enable_icache();
   #endif

   #if XPAR_MICROBLAZE_0_USE_DCACHE
    microblaze_init_dcache_range(0,
XPAR_MICROBLAZE_0_DCACHE_BYTE_SIZE);
    microblaze_enable_dcache();
   #endif

   // Habilitar las interrupciones del EMULADOR
   EMULADOR_EnableInterrupt((void *)XPAR_EMULADOR_0_BASEADDR);

microblaze_register_handler((XInterruptHandler)EMULADOR_Intr_Def
aultHandler, (void *)XPAR_EMULADOR_0_BASEADDR);

   // Enable MicroBlaze interrupts
   microblaze_enable_interrupts();

   xil_printf("\r\n Waiting reception from USB\r\n");

  // Bucle infinito principal del programa
  while(1){
   //Set the endpoint address
   preg->addr = ADDR_EP2; //EP2
   //Start the RX state machine
   preg->ctrl = CTRL_RX_FIFO_EN;
   //Wait until RX FIFO has data (not EMPTY)
   flags = preg->flag;
   while (flags & FLAG_RX_FIFO_EMPTY){
   if (LastEmuladorExpired != EmuladorExpired){
    ResultadosDeEmulacion(registros);
    LastEmuladorExpired = EmuladorExpired;
    NuevoFF = 0xFFFF;
   }
   flags = preg->flag;
   }
   // Read header to obtain instructions from PC
   for(i = 0; i < 6; i++){
    instrucciones[i] = preg->rx_data;      // Read to determine
transfer type  and size
    xil_printf ("\r\n | Instruction %d | %08x |\r\n", i,
instrucciones[i]);
   }
   if (instrucciones[0] == 0xFF) {       // FPGA read
    flags = preg->flag;
    //Discard RX_FIFO contents (only xfer_size required)
    while (!(flags & FLAG_RX_FIFO_EMPTY)){
     data_received = preg->rx_data;
     flags = preg->flag;
    }
    //Disable the RX state machine
    preg->ctrl = !CTRL_RX_FIFO_EN;
```

```
    endp_num = ADDR_EP6; //EP6

    //if(Peticiones == 0x00)
    inicia_emulacion();
    Peticiones = 0x01;
    ResultadosDeEmulacion(registros);

    INJFF = EMULADOR_mReadSlaveReg16(XPAR_EMULADOR_0_BASEADDR);
    FFEND = EMULADOR_mReadSlaveReg3(XPAR_EMULADOR_0_BASEADDR);

    xil_printf ("\r\n | INJFF = %08x | FFEND = %08x |\r\n",
INJFF,  FFEND);

    if(INJFF == FFEND){
     FinEmulacion = 0xFF;
     xil_printf ("\r\nFinished Emulation\r\n");
    }

    usb_tx_results(endp_num, instrucciones, registros,
FinEmulacion);
   }
   else if (instrucciones[0] == 0xFE) {  // Waiting parameters
of Emulation
    endp_num = ADDR_EP2; //EP2
    usb_rx_param(endp_num, instrucciones, registros);

    LecturaDeRegistros();

   }
   else if (instrucciones[0] == 0xFD) {  // Test USB
communication
    flags = preg->flag;
    //Discard RX_FIFO contents (only xfer_size required)
    while (!(flags & FLAG_RX_FIFO_EMPTY)){
     data_received = preg->rx_data;
     flags = preg->flag;
    }
    //Disable the RX state machine
    preg->ctrl = !CTRL_RX_FIFO_EN;
    endp_num = ADDR_EP6; //EP6
    xil_printf ("\r\nStarting Test\r\n");
    USB_Test(endp_num);
   }
   else if (instrucciones[0] == 0xFC) {       // Register clear
    flags = preg->flag;
    //Discard RX_FIFO contents (only xfer_size required)
    while (!(flags & FLAG_RX_FIFO_EMPTY)){
     data_received = preg->rx_data;
     flags = preg->flag;
    }
    //Disable the RX state machine
    preg->ctrl = !CTRL_RX_FIFO_EN;
    endp_num = ADDR_EP6; //EP6

    // Clear registers
```

```
    EMULADOR_mWriteSlaveReg0(XPAR_EMULADOR_0_BASEADDR,
0x00000000);
    EMULADOR_mWriteSlaveReg0(XPAR_EMULADOR_0_BASEADDR,
0x00000004);
    EMULADOR_mWriteSlaveReg0(XPAR_EMULADOR_0_BASEADDR,
0x00000000);
    xil_printf ("\r\nRegister clear done\r\n");

    ResultadosDeEmulacion(registros);

    usb_tx_results(endp_num, instrucciones, registros,
FinEmulacion);
   }
   if (instrucciones[0] == 0xFB) {       // Show last result
    flags = preg->flag;
    //Discard RX_FIFO contents (only xfer_size required)
    while (!(flags & FLAG_RX_FIFO_EMPTY)){
     data_received = preg->rx_data;
     flags = preg->flag;
    }
    //Disable the RX state machine
    preg->ctrl = !CTRL_RX_FIFO_EN;
    endp_num = ADDR_EP6; //EP6

    // Update register data
    EMULADOR_mWriteSlaveReg0(XPAR_EMULADOR_0_BASEADDR,
0x00000000);
    EMULADOR_mWriteSlaveReg0(XPAR_EMULADOR_0_BASEADDR,
0x00000008);
    EMULADOR_mWriteSlaveReg0(XPAR_EMULADOR_0_BASEADDR,
0x00000000);
    xil_printf ("\r\nLast results showed\r\n");

    ResultadosDeEmulacion(registros);

    usb_tx_results(endp_num, instrucciones, registros,
FinEmulacion);
   }
  }
  //Disable the RX state machine
  preg->ctrl = !CTRL_RX_FIFO_EN;

  return 0;
}
```

Función LecturaDeRegistros

```
//Lectura de registros por pantalla, a traves del puerto serie
void LecturaDeRegistros(void)
{

  Xuint32 temp;

  temp = EMULADOR_mReadSlaveReg0(XPAR_EMULADOR_0_BASEADDR);
```

```
  xil_printf ("\r\nRegister 0:  %08x ", temp);
  temp = EMULADOR_mReadSlaveReg1(XPAR_EMULADOR_0_BASEADDR);
  xil_printf ("\r\nRegister 1:  %08x ", temp);
  temp = EMULADOR_mReadSlaveReg2(XPAR_EMULADOR_0_BASEADDR);
  xil_printf ("\r\nRegister 2:  %08x ", temp);
  temp = EMULADOR_mReadSlaveReg3(XPAR_EMULADOR_0_BASEADDR);
  xil_printf ("\r\nRegister 3:  %08x ", temp);
  temp = EMULADOR_mReadSlaveReg4(XPAR_EMULADOR_0_BASEADDR);
  xil_printf ("\r\nRegister 4:  %08x ", temp);
  temp = EMULADOR_mReadSlaveReg5(XPAR_EMULADOR_0_BASEADDR);
  xil_printf ("\r\nRegister 5:  %08x ", temp);
  temp = EMULADOR_mReadSlaveReg6(XPAR_EMULADOR_0_BASEADDR);
  xil_printf ("\r\nRegister 6:  %08x ", temp);
  temp = EMULADOR_mReadSlaveReg7(XPAR_EMULADOR_0_BASEADDR);
  xil_printf ("\r\nRegister 7:  %08x ", temp);
  temp = EMULADOR_mReadSlaveReg8(XPAR_EMULADOR_0_BASEADDR);
  xil_printf ("\r\nRegister 8:  %08x ", temp);
  temp = EMULADOR_mReadSlaveReg9(XPAR_EMULADOR_0_BASEADDR);
  xil_printf ("\r\nRegister 9:  %08x ", temp);
  temp = EMULADOR_mReadSlaveReg10(XPAR_EMULADOR_0_BASEADDR);
  xil_printf ("\r\nRegister 10: %08x ", temp);
  temp = EMULADOR_mReadSlaveReg11(XPAR_EMULADOR_0_BASEADDR);
  xil_printf ("\r\nRegister 11: %08x ", temp);
  temp = EMULADOR_mReadSlaveReg12(XPAR_EMULADOR_0_BASEADDR);
  xil_printf ("\r\nRegister 12: %08x ", temp);
  temp = EMULADOR_mReadSlaveReg13(XPAR_EMULADOR_0_BASEADDR);
  xil_printf ("\r\nRegister 13: %08x ", temp);
  temp = EMULADOR_mReadSlaveReg14(XPAR_EMULADOR_0_BASEADDR);
  xil_printf ("\r\nRegister 14: %08x ", temp);
  temp = EMULADOR_mReadSlaveReg15(XPAR_EMULADOR_0_BASEADDR);
  xil_printf ("\r\nRegister 15: %08x ", temp);
  temp = EMULADOR_mReadSlaveReg16(XPAR_EMULADOR_0_BASEADDR);
  xil_printf ("\r\nRegister 16: %08x ", temp);
  temp = EMULADOR_mReadSlaveReg17(XPAR_EMULADOR_0_BASEADDR);
  xil_printf ("\r\nRegister 17: %08x ", temp);
  temp = EMULADOR_mReadSlaveReg18(XPAR_EMULADOR_0_BASEADDR);
  xil_printf ("\r\nRegister 18: %08x ", temp);
  temp = EMULADOR_mReadSlaveReg19(XPAR_EMULADOR_0_BASEADDR);
  xil_printf ("\r\nRegister 19: %08x ", temp);
  temp = EMULADOR_mReadSlaveReg20(XPAR_EMULADOR_0_BASEADDR);
  xil_printf ("\r\nRegister 20: %08x ", temp);
  temp = EMULADOR_mReadSlaveReg21(XPAR_EMULADOR_0_BASEADDR);
  xil_printf ("\r\nRegister 21: %08x ", temp);
  temp = EMULADOR_mReadSlaveReg22(XPAR_EMULADOR_0_BASEADDR);
  xil_printf ("\r\nRegister 22: %08x ", temp);
  temp = EMULADOR_mReadSlaveReg23(XPAR_EMULADOR_0_BASEADDR);
  xil_printf ("\r\nRegister 23: %08x ", temp);

}
```

Función ResultadosDeEmulación

```
// Function to store Emulation parameters from Emulator
registers
```

```
void ResultadosDeEmulacion(Xuint16 *registros)
{

 Xuint32 temp;
 Xuint16 result;

 temp = EMULADOR_mReadSlaveReg0(XPAR_EMULADOR_0_BASEADDR);
 registros[0] = temp >> 16;
 registros[1] = temp;
 temp = EMULADOR_mReadSlaveReg1(XPAR_EMULADOR_0_BASEADDR);
 registros[2] = temp >> 16;
 registros[3] = temp;
 temp = EMULADOR_mReadSlaveReg16(XPAR_EMULADOR_0_BASEADDR);
 registros[32] = temp >> 16;
 registros[33] = temp;
 temp = EMULADOR_mReadSlaveReg17(XPAR_EMULADOR_0_BASEADDR);
 registros[34] = temp >> 16;
 registros[35] = temp;
 temp = EMULADOR_mReadSlaveReg18(XPAR_EMULADOR_0_BASEADDR);
 registros[36] = temp >> 16;
 registros[37] = temp;
 temp = EMULADOR_mReadSlaveReg19(XPAR_EMULADOR_0_BASEADDR);
 registros[38] = temp >> 16;
 registros[39] = temp;
 temp = EMULADOR_mReadSlaveReg20(XPAR_EMULADOR_0_BASEADDR);
 registros[40] = temp >> 16;
 registros[41] = temp;
 temp = EMULADOR_mReadSlaveReg21(XPAR_EMULADOR_0_BASEADDR);
 registros[42] = temp >> 16;
 registros[43] = temp;
 temp = EMULADOR_mReadSlaveReg22(XPAR_EMULADOR_0_BASEADDR);
 registros[44] = temp >> 16;
 registros[45] = temp;
 temp = EMULADOR_mReadSlaveReg23(XPAR_EMULADOR_0_BASEADDR);
 registros[46] = temp >> 16;
 registros[47] = temp;

}
```

Función ParametrosDeEmulacion

```
// Function to store Emulation parameters at Emulator registers
void ParametrosDeEmulacion(Xuint16 *registros)
{

  Xuint32 temp;

  xil_printf("\r\n Load parameters in Emulator registers ");

  temp = registros[0];
  temp = (temp << 16) + registros[1];
  EMULADOR_mWriteSlaveReg0(XPAR_EMULADOR_0_BASEADDR, temp);
  temp = registros[2];
  temp = (temp << 16) + registros[3];
```

```
EMULADOR_mWriteSlaveReg1(XPAR_EMULADOR_0_BASEADDR, temp);
temp = registros[4];
temp = (temp << 16) + registros[5];
EMULADOR_mWriteSlaveReg2(XPAR_EMULADOR_0_BASEADDR, temp);
temp = registros[6];
temp = (temp << 16) + registros[7];
temp++;
EMULADOR_mWriteSlaveReg3(XPAR_EMULADOR_0_BASEADDR, temp);
temp = registros[8];
temp = (temp << 16) + registros[9];
EMULADOR_mWriteSlaveReg4(XPAR_EMULADOR_0_BASEADDR, temp);
temp = registros[10];
temp = (temp << 16) + registros[11];
EMULADOR_mWriteSlaveReg5(XPAR_EMULADOR_0_BASEADDR, temp);
temp = registros[12];
temp = (temp << 16) + registros[13];
EMULADOR_mWriteSlaveReg6(XPAR_EMULADOR_0_BASEADDR, temp);
temp = registros[14];
temp = (temp << 16) + registros[15];
EMULADOR_mWriteSlaveReg7(XPAR_EMULADOR_0_BASEADDR, temp);
temp = registros[16];
temp = (temp << 16) + registros[17];
EMULADOR_mWriteSlaveReg8(XPAR_EMULADOR_0_BASEADDR, temp);
temp = registros[18];
temp = (temp << 16) + registros[19];
EMULADOR_mWriteSlaveReg9(XPAR_EMULADOR_0_BASEADDR, temp);
temp = registros[20];
temp = (temp << 16) + registros[21];
EMULADOR_mWriteSlaveReg10(XPAR_EMULADOR_0_BASEADDR, temp);
temp = registros[22];
temp = (temp << 16) + registros[23];
EMULADOR_mWriteSlaveReg11(XPAR_EMULADOR_0_BASEADDR, temp);
temp = registros[24];
temp = (temp << 16) + registros[25];
EMULADOR_mWriteSlaveReg12(XPAR_EMULADOR_0_BASEADDR, temp);
temp = registros[26];
temp = (temp << 16) + registros[27];
EMULADOR_mWriteSlaveReg13(XPAR_EMULADOR_0_BASEADDR, temp);
temp = registros[28];
temp = (temp << 16) + registros[29];
EMULADOR_mWriteSlaveReg14(XPAR_EMULADOR_0_BASEADDR, temp);
temp = registros[30];
temp = (temp << 16) + registros[31];
EMULADOR_mWriteSlaveReg15(XPAR_EMULADOR_0_BASEADDR, temp);
temp = registros[32];
temp = (temp << 16) + registros[33];
EMULADOR_mWriteSlaveReg16(XPAR_EMULADOR_0_BASEADDR, temp);
temp = registros[34];
temp = (temp << 16) + registros[35];
EMULADOR_mWriteSlaveReg17(XPAR_EMULADOR_0_BASEADDR, temp);
temp = registros[36];
temp = (temp << 16) + registros[37];
EMULADOR_mWriteSlaveReg18(XPAR_EMULADOR_0_BASEADDR, temp);
temp = registros[38];
temp = (temp << 16) + registros[39];
EMULADOR_mWriteSlaveReg19(XPAR_EMULADOR_0_BASEADDR, temp);
```

```
  temp = registros[40];
  temp = (temp << 16) + registros[41];
  EMULADOR_mWriteSlaveReg20(XPAR_EMULADOR_0_BASEADDR, temp);
  temp = registros[42];
  temp = (temp << 16) + registros[43];
  EMULADOR_mWriteSlaveReg21(XPAR_EMULADOR_0_BASEADDR, temp);
  temp = registros[44];
  temp = (temp << 16) + registros[45];
  EMULADOR_mWriteSlaveReg22(XPAR_EMULADOR_0_BASEADDR, temp);
  temp = registros[46];
  temp = (temp << 16) + registros[47];
  EMULADOR_mWriteSlaveReg23(XPAR_EMULADOR_0_BASEADDR, temp);

}
```

Función usb_rx_param

```
void usb_rx_param(Xuint8 endp_num, Xuint16 *instrucciones,
Xuint16 *registros)
{

  Xuint8 flags;
  Xuint16 i, temp;

  //Set the endpoint address
  preg->addr = endp_num;
  //Start the RX state machine
  preg->ctrl = CTRL_RX_FIFO_EN;
  flags = preg->flag;

  i = 0;
  while (!(flags & FLAG_RX_FIFO_EMPTY)){
   temp = preg->rx_data;
   registros[i] = temp;
   flags = preg->flag;
   i++;
  }
  //Disable the RX state machine
  preg->ctrl = !CTRL_RX_FIFO_EN;

  ParametrosDeEmulacion(registros);

  return;

}
```

Función usb_tx_results

```
void usb_tx_results(Xuint8 endp_num, Xuint16 *instrucciones,
Xuint16 *registros, Xuint8 FinEmulacion)
{

  Xuint8 flags;
```

```
  Xuint16 errorCount, xfer_req_LW, Continuar, temp;
  Xuint32 i, countL;

  if (FinEmulacion == 0x00)
   Continuar = 0xFFFF;
  else
   Continuar = 0x0000;

  // N° de bytes pedidos => se envian 2 bytes por slot del
buffer de salida
  xfer_req_LW = 0x0100;
  //Set the endpoint address
  preg->addr = endp_num;

  flags = preg->flag;
  while (flags & FLAG_ENDP_FULL) {
   print(":-(");
  }

  //Start the TX state machine
  preg->ctrl = CTRL_TX_FIFO_EN;

  for (countL=0; countL<=0xFFFF; countL++) {
   // Send data
   errorCount=0;
   do {
    flags = preg->flag;
     errorCount++;
   } while ((flags & FLAG_ENDP_FULL)&&(errorCount<10000));

   if (errorCount==10000) {
    break;
   }
   else {
    if(countL == 0x0000){
     preg->tx_data = Continuar;
     i = 0;
    }
    else if (countL == 0x0001)
     preg->tx_data = NuevoFF;
    else if (countL < 0x0034){
     temp = registros[i-2];
     preg->tx_data = temp;
     i++;
    }
    else
     preg->tx_data = countL;

      // Exit if count limit reached
      if (countL==(xfer_req_LW-1))
       break;
   }
   if (NuevoFF == 0xFFFF)
     NuevoFF = 0x0000;

   xil_printf("\r\nValor de instruccion: %08x", Continuar);
```

```
   //Wait until all data in FIFO xfer to ENDPT
   flags = preg->flag;
   while (!(flags & FLAG_TX_FIFO_EMPTY)){
      flags = preg->flag;
   }

  //Disable the TX state machine
  preg->ctrl = !CTRL_TX_FIFO_EN;

}
```

Función inicia_emulacion

```
void inicia_emulacion(void)
{

 // Flanco de subida para poner en modo Run el emulador
 EMULADOR_mWriteSlaveReg0(XPAR_EMULADOR_0_BASEADDR, 0x00000000);
 EMULADOR_mWriteSlaveReg0(XPAR_EMULADOR_0_BASEADDR, 0x00000001);
 xil_printf ("\r\nEmulation started\r\n");

}
```

Función EMULADOR_EnableInterrupt

```
void EMULADOR_EnableInterrupt(void * baseaddr_p)
{

  Xuint32 baseaddr;
  baseaddr = (Xuint32) baseaddr_p;

  // Enable all interrupt source from user logic.
  EMULADOR_mWriteReg(baseaddr, EMULADOR_INTR_IER_OFFSET,
0x00000001);

  // Enable all possible interrupt sources from device.
  EMULADOR_mWriteReg(baseaddr, EMULADOR_INTR_DIER_OFFSET,
    INTR_TERR_MASK
    | INTR_DPTO_MASK
    | INTR_IPIR_MASK
    );

  //Set global interrupt enable.
  EMULADOR_mWriteReg(baseaddr, EMULADOR_INTR_DGIER_OFFSET,
INTR_GIE_MASK);

}
```

Función EMULADOR_Intr_DefaultHandler

```
// Interrupt controller handler for EMULADOR device.
void EMULADOR_Intr_DefaultHandler(void * baseaddr_p)
{

 Xuint32 baseaddr;
 Xuint32 IntrStatus;
 Xuint32 IpStatus;

 baseaddr = (Xuint32) baseaddr_p;

 EmuladorExpired++;
 // Get status from Device Interrupt Status Register.
 IntrStatus = EMULADOR_mReadReg(baseaddr,
EMULADOR_INTR_DISR_OFFSET);

 // Verify the source of the interrupt is the user logic and
clear the interrupt
 // source by toggle write baca to the IP ISR register.

 if ( (IntrStatus & INTR_IPIR_MASK) == INTR_IPIR_MASK )
 {
  IpStatus = EMULADOR_mReadReg(baseaddr,
EMULADOR_INTR_ISR_OFFSET);
  EMULADOR_mWriteReg(baseaddr, EMULADOR_INTR_ISR_OFFSET,
IpStatus);
 }

}
```

Función USB_Test

```
void USB_Test(Xuint8 endp_num)
{

  Xuint8 flags;
  Xuint16 errorCount, xfer_req_LW, temp;
  Xuint32 i, countL;


  // N° de bytes pedidos => se envian 2 bytes por slot del
buffer de salida
  xfer_req_LW = 0x0100;
  //Set the endpoint address
  preg->addr = endp_num;

   flags = preg->flag;
  while (flags & FLAG_ENDP_FULL) {
   print(":-(");
  }

  //Start the TX state machine
```

```
  preg->ctrl = CTRL_TX_FIFO_EN;

  for (countL=0; countL<=0xFFFF; countL++) {
   // Send data
   errorCount=0;
   do {
    flags = preg->flag;
    errorCount++;
   } while ((flags & FLAG_ENDP_FULL)&&(errorCount<10000));

   if (errorCount==10000) {
    break;
   }
   else {
    if(countL == 0x0000){
    preg->tx_data = countL;
    i = 0;
    }
     else if (countL == 0x0001)
      preg->tx_data = countL;
    else if (countL < 0x0032){
      preg->tx_data = countL;
      i++;
     }
    else
     preg->tx_data = countL;

   // Exit if count limit reached
   if (countL==(xfer_req_LW-1))
    break;
   }
  }

  //Wait until all data in FIFO xfer to ENDPT
  flags = preg->flag;
  while (!(flags & FLAG_TX_FIFO_EMPTY)){
   flags = preg->flag;
  }

  //Disable the TX state machine
  preg->ctrl = !CTRL_TX_FIFO_EN;

  xil_printf ("\r\nTest finished\r\n");

}
```

ANEXO II: CÓDIGO DE LAS FUNCIONES DE LA INTERFAZ DE USUARIO

Bibliotecas USB2UtilDlg

```
#include "stdafx.h"
#include "USB2Util.h"
#include "USB2UtilDlg.h"
#include <process.h>
#include <winioctl.h>
#include <windows.h>
#include <stdio.h>
#include <Afxcmn.h>
#include "Results.h"
```

Función OnInitDialog

```
BOOL CUSB2UtilDlg::OnInitDialog()
{
  CDialog::OnInitDialog();

  // Add "About..." menu item to system menu.

  // IDM_ABOUTBOX must be in the system command range.
  ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
  ASSERT(IDM_ABOUTBOX < 0xF000);

  CMenu* pSysMenu = GetSystemMenu(FALSE);
  if (pSysMenu != NULL)
 {
   CString strAboutMenu;
   strAboutMenu.LoadString(IDS_ABOUTBOX);
   if (!strAboutMenu.IsEmpty())
   {
     pSysMenu->AppendMenu(MF_SEPARATOR);
     pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX,
strAboutMenu);
   }
  }

  // Set the icon for this dialog.  The framework does this
automatically
  //  when the application's main window is not a dialog
  SetIcon(m_hIcon, TRUE);            // Set big icon
  SetIcon(m_hIcon, FALSE);      // Set small icon

  m_tbCtrl.InitDialogs();

  m_tbCtrl.InsertItem(0,"Configuration");
  m_tbCtrl.InsertItem(1,"Results");
```

```
  m_tbCtrl.ActivateTabDialogs();
  // Show ADS Banner in Lower Right corner
  m_strADSBitmap = "ADS Banner.bmp";
  HBITMAP hADSBitmap = (HBITMAP)
::LoadImage(AfxGetInstanceHandle(),

m_strADSBitmap, IMAGE_BITMAP, 0, 0,

LR_LOADFROMFILE | LR_CREATEDIBSECTION);
  if(hADSBitmap)
  {
   if(m_bmpADSBitmap.DeleteObject())
   {
    m_bmpADSBitmap.Detach();
    m_bmpADSBitmap.Attach(hADSBitmap);
   }
  }
   UpdateData(FALSE);

   return TRUE;  // return TRUE  unless you set the focus to a
control
}
```

Bibliotecas y constantes de Configuration

```
#include "stdafx.h"
#include "USB2Util.h"
#include "Configuration.h"

#define    FILE_CONFIG          0
#define    FILE_PARAM           1
#define    FILE_RESULTS         2

#define    BD_VIRTEX_4_EVAL     0
#define    BD_VIRTEX_5_XUP      1
define     BD_GENERIC           2

#define    V4EVAL_BOARD_ID_ADDR 0x0ffc
#define    V5XUP_BOARD_ID_ADDR  0x0ffc
```

Función CConfiguration

```
CConfiguration::CConfiguration(CWnd* pParent /*=NULL*/)
     : CDialog(CConfiguration::IDD, pParent)
{

  m_intCntlMode = 0;
  m_strCntlFilename = _T("");
  m_strCntlConfigurationFile = _T("Configuration");
  m_strCntlParametersFile = _T("Parameters");
  m_strCntlResultsFile = _T("Results");
```

```
}
```

Función DoDataExchange

```
void CConfiguration::DoDataExchange(CDataExchange* pDX)
{

  CDialog::DoDataExchange(pDX);
  DDX_Text(pDX, IDC_CONFIGURATION_FILE,
m_strCntlConfigurationFile);
  DDX_Text(pDX, IDC_PARAMETERS_FILE, m_strCntlParametersFile);
  DDX_Text(pDX, IDC_RESULTS_FILE, m_strCntlResultsFile);
  DDX_CBIndex(pDX, IDC_DEVICE_TYPE, m_intBoard);

}
```

Mapa de mensajes de Configuration

```
BEGIN_MESSAGE_MAP(CConfiguration, CDialog)
  ON_CBN_SELCHANGE(IDC_DEVICE_TYPE,
&CConfiguration::OnCbnSelchangeDeviceType)
  ON_BN_CLICKED(IDC_BROWSE_CONFIG_FILE,
&CConfiguration::OnBnClickedBrowseConfigFile)
  ON_BN_CLICKED(IDC_BROWSE_PARAM_FILE,
&CConfiguration::OnBnClickedBrowseParamFile)
  ON_BN_CLICKED(IDC_BOWSE_SAVE_RESULTS,
&CConfiguration::OnBnClickedBowseSaveResults)
  ON_EN_CHANGE(IDC_CONFIGURATION_FILE,
&CConfiguration::OnEnChangeConfigurationFile)
  ON_EN_CHANGE(IDC_PARAMETERS_FILE,
&CConfiguration::OnEnChangeParametersFile)
  ON_EN_CHANGE(IDC_RESULTS_FILE,
&CConfiguration::OnEnChangeResultsFile)
END_MESSAGE_MAP()
```

Función GetFileName

```
BOOLEAN CConfiguration::GetFileName()
{

  UpdateData(TRUE);
  CString args;
  switch(m_intCntlFile)
  {
   case FILE_CONFIG:
    args = "Configuration file (*.hex)|*.hex|";
    break;
   case FILE_PARAM:
    args = "Parameters file (*.ini)|*.ini|";
    break;
   case FILE_RESULTS:
```

```
    args = "Results File (*.res)|*.res|";
    break;
   default:
    args = "All files (*.*)|*.*||";
    break;
  }

  CFileDialog m_ldFile(TRUE, NULL, NULL, OFN_OVERWRITEPROMPT,
args);
  // Select a file and get its name
  if(m_ldFile.DoModal() == IDOK)
  {
    m_strCntlFilename = m_ldFile.GetFileName();
    m_strPathName = m_ldFile.GetPathName();
  }

  return FALSE;
}
```

Función OnCbnSelchangeDeviceType

```
void CConfiguration::OnCbnSelchangeDeviceType()
{

  UpdateData(TRUE);

  m_strConfigFile = " Configuration.cfg";

  switch(m_intBoard)
  {
   case BD_VIRTEX_4_EVAL:        // Virtex 4 Eval
    m_intBoardIDAddr = V4EVAL_BOARD_ID_ADDR;
   break;
   case BD_VIRTEX_5_XUP:         // ADS Video Dev
    m_intBoardIDAddr = V5XUP_BOARD_ID_ADDR;
   break;
   case BD_GENERIC:
    m_intBoardIDAddr = V4EVAL_BOARD_ID_ADDR;
   break;
   default:
    MessageBox("A board has not been chosen.  Please select a
board.");
   break;
  }

  UpdateConfig();

  UpdateData(FALSE);

}
```

Función OnBnClickedBrowseConfigFile

```
void CConfiguration::OnBnClickedBrowseConfigFile()
{

  UpdateData(TRUE);
  m_intCntlFile = FILE_CONFIG;
  GetFileName();

  m_strCntlConfigurationFile = m_strCntlFilename;
  m_strPathConfiguration = m_strPathName;


  UpdateData(FALSE);

}
```

Función OnBnClickedBrowseParamFile

```
void CConfiguration::OnBnClickedBrowseParamFile()
{

  UpdateData(TRUE);
  m_intCntlFile = FILE_PARAM;
  GetFileName();

  m_strCntlParametersFile = m_strCntlFilename;
  m_strPathParameters = m_strPathName;
  UpdateData(FALSE);

}
```

Función OnBnClickedBrowseSaveResults

```
void CConfiguration::OnBnClickedBowseSaveResults()
{

  UpdateData(TRUE);
  m_intCntlFile = FILE_RESULTS;
  GetFileName();

  m_strCntlResultsFile = m_strCntlFilename;
  m_strPathResults = m_strPathName;

UpdateData(FALSE);

}
```

Función UpdateConfig

```
BOOLEAN CConfiguration::UpdateConfig(void)
{

  FILE *ConfigFile;
  int FileColumn = 0;
  int FileRow = 0;
  char FileContent[100][33];          // File dimension
  int ColumnSize = 0;
  int RowSize = 0;
  BOOLEAN FIN = FALSE;
  int i = 0;
  int j = 0;

  for (i = 0; i < 100; i++){
   for (j = 0; j < 33; j++)
    FileContent[i][j] = ' ';
  }

  if((ConfigFile = fopen(m_strConfigFile, "r")) == NULL)
    MessageBox("Couldn't open file for reading");
  else{
   rewind(ConfigFile);
   // Safe File content
   while(!feof(ConfigFile)){
    FileContent[FileRow][FileColumn] = fgetc(ConfigFile);
    if(FileContent[FileRow][FileColumn] == '\n'){
     FileRow++;
     FileColumn = 0;
    }
    else
     FileColumn++;
    }

    rewind(ConfigFile);
    fclose(ConfigFile);
   }
   RowSize = FileRow;
   ColumnSize = FileColumn;
   // Modify variables
   FileRow = 0;
   FileColumn = 0;
   if((ConfigFile = fopen(m_strConfigFile, "w")) == NULL)
    MessageBox("Couldn't open file for writing");
   else{
    rewind(ConfigFile);
    SetCurrentDate(ConfigFile);
    FileRow++;
    // Safe File content
    while(FIN == FALSE){
     if(FileRow == 4){
      FileRow++;
      FileColumn = 0;
```

```
      fprintf(ConfigFile, "CfgFile = %s\n",
m_strPathConfiguration);
     }
     else if(FileRow == 5){
      FileRow++;
      FileColumn = 0;
      fprintf(ConfigFile, "ParFile = %s\n",
m_strPathParameters);
     }
     else if(FileRow == 6){
      FileRow++;
      FileColumn = 0;
      fprintf(ConfigFile, "ParFile = %s\n", m_strPathResults);
     }
     else if(FileRow == 8){
      FileRow++;
      FileColumn = 0;
      switch(m_intBoard){
       case BD_VIRTEX_4_EVAL:         // Virtex 4 Eval
        fputs("Dev = V4 Evaluation Board\n", ConfigFile);
       break;
      case BD_VIRTEX_5_XUP:           // ADS Video Dev
       fputs("Dev = V5 Xilinx University Program\n",
ConfigFile);
      break;
      case BD_GENERIC:
       fputs("Dev = Generic\n", ConfigFile);
      break;
      default:
      break;
      }
     }
     else{
      fputc(FileContent[FileRow][FileColumn], ConfigFile);
      if(FileContent[FileRow][FileColumn] == '\n'){
       FileRow++;
       FileColumn = 0;
      }
      else
       FileColumn++;
     }
     if(FileRow == (RowSize-1)){
      FIN = TRUE;
      fputs("\n#End of File", ConfigFile);
     }
    }
    rewind(ConfigFile);
    fclose(ConfigFile);
   }

  return TRUE;

}
```

Función SetCurrentDate

```
void CConfiguration::SetCurrentDate(FILE *CurrentFile)
{

  WORD Year      = 0;
  WORD Month     = 0;
  WORD DayOfWeek = 0;
  WORD Day       = 0;
  WORD Hour      = 0;
  WORD Minute    = 0;
  WORD Second    = 0;

  SYSTEMTIME CurrentTime;

  GetLocalTime(&CurrentTime);

  Year = CurrentTime.wYear;
  Month = CurrentTime.wMonth;
  DayOfWeek = CurrentTime.wDayOfWeek;

  Hour = CurrentTime.wHour;
  Minute = CurrentTime.wMinute;
  Second = CurrentTime.wSecond;

  fprintf(CurrentFile, ";Last modified: ");
  IntDayOfWeek2File(DayOfWeek, CurrentFile);
  IntMonth2File(Month, CurrentFile);
  DayOfMonth2File(Day, CurrentFile);
  fprintf(CurrentFile, "%d", Year);
  fprintf(CurrentFile, " at ");
  fprintf(CurrentFile, "%02d", Hour);
  fprintf(CurrentFile, ":");
  fprintf(CurrentFile, "%02d", Minute);
  fprintf(CurrentFile, ":");
  fprintf(CurrentFile, "%02d\n", Second);

}
```

Función IntMonth2File

```
void CConfiguration::IntMonth2File(WORD Month, FILE
*CurrentFile)
{

  switch (Month){
   case 1:
    fprintf(CurrentFile, "January ");
   break;
   case 2:
    fprintf(CurrentFile, "February ");
   break;
   case 3:
```

```
    fprintf(CurrentFile, "March ");
   break;
   case 4:
    fprintf(CurrentFile, "April ");
   break;
   case 5:
    fprintf(CurrentFile, "May ");
   break;
   case 6:
    fprintf(CurrentFile, "June ");
   break;
   case 7:
    fprintf(CurrentFile, "July ");
   break;
   case 8:
    fprintf(CurrentFile, "August ");
   break;
   case 9:
    fprintf(CurrentFile, "September ");
   break;
   case 10:
    fprintf(CurrentFile, "October ");
   break;
   case 11:
    fprintf(CurrentFile, "November ");
   break;
   case 12:
    fprintf(CurrentFile, "December ");
   break;
   default:
   break;
  }

}
```

Función IntDayOfWeek2File

```
void CConfiguration::IntDayOfWeek2File(WORD DayOfWeek, FILE
*CurrentFile)
{

  switch (DayOfWeek){
   case 1:
    fprintf(CurrentFile, "Monday, ");
   break;
   case 2:
    fprintf(CurrentFile, "Tuesday, ");
   break;
   case 3:
    fprintf(CurrentFile, "Wednesday, ");
   break;
   case 4:
    fprintf(CurrentFile, "Thursday, ");
   break;
```

```
   case 5:
    fprintf(CurrentFile, "Friday, ");
   break;
   case 6:
    fprintf(CurrentFile, "Saturday, ");
   break;
   case 7:
    fprintf(CurrentFile, "Sunday, ");
   break;
   default:
   break;
  }

}
```

Función DayOfMonth2File

```
void CConfiguration::DayOfMonth2File(WORD Day, FILE
*CurrentFile)
{

  SYSTEMTIME CurrentTime;

  GetLocalTime(&CurrentTime);

  fprintf(CurrentFile, "%2d", CurrentTime.wDay);
  switch (Day){
   case 1:
    fprintf(CurrentFile, "st, ");
   break;
   case 2:
    fprintf(CurrentFile, "nd, ");
   break;
   case 3:
    fprintf(CurrentFile, "rd, ");
   break;
   case 21:
    fprintf(CurrentFile, "st, ");
   break;
   case 22:
    fprintf(CurrentFile, "nd, ");
   break;
   case 23:
    fprintf(CurrentFile, "rd, ");
   break;
   case 31:
    fprintf(CurrentFile, "st, ");
   break;
   default:
    fprintf(CurrentFile, "th, ");
   break;
  }

}
```

Bibliotecas y constantes de Results

```
#include "stdafx.h"
#include "USB2Util.h"
#include "USB2UtilDlg.h"
#include <process.h>
#include <winioctl.h>
#include <windows.h>
#include <stdio.h>
#include <Afxcmn.h>
#include "Results.h"

//   Commands sent to board
#define    READ                  0xff
#define    WRITE                 0xfe
#define    TEST                  0xfd
#define    CLEAR                 0xfc
#define    SHOW                  0xfb
#define    RESET                 0xfa
#define    ID_SET                0xf9
#define    ID_GET                0xf8

#define    SIZE_OF_HEADER        12
#define    MIN_RETURN_BYTES4

#define    MD_READ               0
#define    MD_WRITE              1
#define    MD_CONFIG_FX2         2
#define    MD_CONFIG_FPGA        3
#define    MD_SET_BOARD_ID       4
#define    MD_GET_BOARD_ID       5

#define    _1K                   1024
#define    _2K                   (2 * _1K)
#define    _3K                   (3 * _1K)
#define    _4K                   (4 * _1K)
#define    _5K                   (5 * _1K)
#define    _6K                   (6 * _1K)
#define    _7K                   (7 * _1K)
#define    _8K                   (8 * _1K)
#define _512K                    (512 * _1K)

#define    V4EVAL_BOARD_ID_ADDR  0x0ffc
#define    V5XUP_BOARD_ID_ADDR   0x1ffc
#define    GENERIC_BOARD_ID_ADDR 0x1ffc

#define    PP                    0x02  // Page Program command
#define    BE                    0xc7  // Bulk Erase command
#define    SE                    0xd8  // Sector Erase command

#define MAX_DEVICES              10

#define    AVNET_VENDOR_ID       0x01000000
#define    V4EVAL_BOARD_ID       0x00200000
#define    V4_LX60               0x00000200
```

Función DoDataExchange

```
void CResults::DoDataExchange(CDataExchange* pDX)
{

  CDialog::DoDataExchange(pDX);
  DDX_Control(pDX, IDC_LIST_RESULTS, m_listResults);
  DDX_Control(pDX, IDC_PROGRESS_BAR, m_cntlProgressBar);
  DDX_Control(pDX, IDC_STATIC_STATUS, m_cntlStatus);
  DDX_Control(pDX, IDC_STATIC_CONECTADO_LED_OFF,
m_ConnectedLedOff);
  DDX_Control(pDX, IDC_STATIC_COMUNICANDO_LED_OFF,
m_TransferingLedOff);
  DDX_Control(pDX, IDC_STATIC_CONECTADO_LED_ON,
m_ConnectedLedOn);
  DDX_Control(pDX, IDC_STATIC_COMUNICANDO_LED_ON,
m_TransferingLedOn);
  DDX_Control(pDX, IDC_STATIC_DIRECTION, m_cntlDirection);
  DDX_Control(pDX, IDC_CURRENT_FF, m_cntlCurrentFF);
  DDX_Control(pDX, IDC_CURRENT_CYCLE, m_cntlCurrentCycle);
  DDX_Control(pDX, IDC_SILENTS, m_cntlSilents);
  DDX_Control(pDX, IDC_FAILURES, m_cntlFailures);
  DDX_Control(pDX, IDC_LATENTS, m_cntlLatents);
  DDX_Control(pDX, IDC_TOTAL, m_cntlTotal);
  DDX_Control(pDX, IDC_ETA_HOUR, m_cntlETAHour);
  DDX_Control(pDX, IDC_ETA_MIN, m_cntlETAMin);
  DDX_Control(pDX, IDC_ETA_SEC, m_cntlETASec);
  DDX_Control(pDX, IDC_ETA_PERCENT, m_cntlETAPercent);
  DDX_Control(pDX, IDC_BTN_CLEAR, m_clear_button);

}
```

Mapa de mensajes de Results

```
BEGIN_MESSAGE_MAP(CResults, CDialog)
      ON_BN_CLICKED(IDC_BUTTON_TEST_COMM,
&CResults::OnBnClickedButtonTestComm)
      ON_BN_CLICKED(IDC_BUTTON_LOAD_PARAM,
&CResults::OnBnClickedButtonLoadParam)
      ON_BN_CLICKED(IDC_BTN_EXECUTE,
&CResults::OnBnClickedBtnExecute)
      ON_BN_CLICKED(IDC_BUTTON_CONFIG_COMM,
&CResults::OnBnClickedButtonConfigComm)
      ON_BN_CLICKED(IDC_BTN_CLEAR,
&CResults::OnBnClickedBtnClear)
      ON_BN_CLICKED(IDC_BTN_SHOW, &CResults::OnBnClickedBtnShow)
      ON_STN_CLICKED(IDC_STATIC_CONECTADO,
&CResults::OnStnClickedStaticConectado)
END_MESSAGE_MAP()
```

Función OnBnClickedButtonConfigComm

```
void CResults::OnBnClickedButtonConfigComm()
{

  // Set direction of data(PC to MB)
  m_cntlDirection.SetWindowTextA("<=====");
  m_intCntlMode = MD_CONFIG_FX2;
  ConfigureUSB(TRUE);

}
```

Función OnBnClickedButtonTestComm

```
void CResults::OnBnClickedButtonTestComm()
{

  if(TestUSBComm() == TRUE)
   MessageBox("USB Test passed correctly.");
  else
   MessageBox("USB Test failed.");

}
```

Función TestUSBComm

```
BOOLEAN CResults::TestUSBComm()
{

  BULK_TRANSFER_CONTROL outBulkControl, inBulkControl;
  THREAD_CONTROL outThreadControl, inThreadControl;
  HANDLE WriteCompleteEvent, ReadCompleteEvent;
  HANDLE hOutDevice, hInDevice;
  ULONG inXferSize, outXferSize;
  ULONG MaxTransferSize;
  ULONG OutPipeNum, InPipeNum;
  PUCHAR outBuffer, inBuffer;
  char DeviceName[MAX_DRIVER_NAME];
  SYSTEMTIME TestUSBTime;
  CString DeviceNumber;

  ULONG i;
  inXferSize = 0x00000200;

  // Get size of file to transfer
  MaxTransferSize = 0;     // Clear the MaxTransferSize

  GetLocalTime(&TestUSBTime);

  DeviceNumber = _T("0"); // Set to 0 value for debug

  sprintf(DeviceName, "EZUSB-" + DeviceNumber);
```

```
  outXferSize = MaxTransferSize + SIZE_OF_HEADER + 4 ;     //
Size of data + Max header

  //  Open device
  if (bOpenDriver (&hOutDevice, DeviceName) != TRUE)
  {
   hOutDevice = NULL;
   m_Connected = 0;
   LEDUpdate();
   return FALSE;
  }
  else{
   m_Connected = 1;
   LEDUpdate();
  }
  if (bOpenDriver(&hInDevice, DeviceName) != TRUE)
  {
   hInDevice = NULL;
   if(hOutDevice)
    CloseHandle(hOutDevice);

   m_Connected = 0;
   LEDUpdate();
   return FALSE;
  }
  else{
   m_Connected = 1;
   LEDUpdate();
  }

  // Setup event object
  WriteCompleteEvent = CreateEvent(0,FALSE,FALSE,NULL);
  ReadCompleteEvent = CreateEvent(0,FALSE,FALSE,NULL);

  // Get pipe number (Default output pipe for ADS Boards)
  OutPipeNum = 0;
  InPipeNum = 2;

  outBuffer = (UCHAR *)malloc(outXferSize);
  inBuffer = (UCHAR *)malloc(inXferSize);

  for(i = 0; i < outXferSize; i++)
   outBuffer[i] = 0;

  for(i = 0; i < inXferSize; i++)
   inBuffer[i] = 0;

  // Load instruction for receive transmision
  // Set Header
  outBuffer[0] = TEST;                // Byte 0  command
  outBuffer[1] = 0x00;                // Byte 1  command
  outBuffer[2] = 0x00;                // Byte 2  transfer size
LSB
  outBuffer[3] = 0x01;                // Byte 3  transfer size
MSB
```

```
  outBuffer[4] = 0x00;                // Byte 4  reserved
  outBuffer[5] = 0x00;                // Byte 5  reserved
  outBuffer[6] = 0x00;                // Byte 6  reserved
  outBuffer[7] = 0x00;                // Byte 7  reserved
  outBuffer[8] = 0x00;                // Byte 8  reserved
  outBuffer[9] = 0x00;                // Byte 9  reserved
  outBuffer[10] = 0x00;               // Byte 10 reserved
  outBuffer[11] = 0x00;               // Byte 11 reserved

  // initialize data structures for the write thread
  outBulkControl.pipeNum = OutPipeNum;
  outThreadControl.hDevice = hOutDevice;
  outThreadControl.Ioctl = IOCTL_EZUSB_BULK_WRITE;
  outThreadControl.InBuffer = (PVOID)&outBulkControl;
  outThreadControl.InBufferSize = sizeof(BULK_TRANSFER_CONTROL);

  outThreadControl.OutBuffer = outBuffer;
  outThreadControl.OutBufferSize = outXferSize;

  outThreadControl.completionEvent = WriteCompleteEvent;
  outThreadControl.status = FALSE;
  outThreadControl.BytesReturned = 0;

  m_Transfering = 1;
  LEDUpdate();

  // Start transfer
  AfxBeginThread((AFX_THREADPROC)TransferThread,
(LPVOID)&outThreadControl);

  // Wait for complete event
  WaitForSingleObject(WriteCompleteEvent,INFINITE);

  //  Check if write failed
  if (!outThreadControl.status)
  {
    if(hOutDevice)
     CloseHandle(hOutDevice);
    if(WriteCompleteEvent)
     CloseHandle(WriteCompleteEvent);
     return FALSE;
   }

  m_Transfering = 0;
  LEDUpdate();

  inXferSize = 0x00000200;

  // initialize data structures for the read thread
  inBulkControl.pipeNum = InPipeNum;
  inThreadControl.hDevice = hInDevice;
  inThreadControl.Ioctl = IOCTL_EZUSB_BULK_READ;

  inThreadControl.InBuffer = (PVOID)&inBulkControl;
  inThreadControl.InBufferSize = sizeof(BULK_TRANSFER_CONTROL);
```

```
  inThreadControl.OutBuffer = &inBuffer[0];
  inThreadControl.OutBufferSize = inXferSize;

  inThreadControl.completionEvent = ReadCompleteEvent;
  inThreadControl.status = FALSE;
  inThreadControl.BytesReturned = 0;

  m_Transfering = 1;
  LEDUpdate();

  // Start transfer
  AfxBeginThread((AFX_THREADPROC)TransferThread,
      (LPVOID)&inThreadControl);

  // Wait for complete event
  WaitForSingleObject(ReadCompleteEvent, INFINITE);

  if (!inThreadControl.status)  // Check if read failed
  {
     DWORD error =  GetLastError();
     if(hInDevice)
      CloseHandle(hInDevice);
     if(ReadCompleteEvent)
      CloseHandle(ReadCompleteEvent);
     return FALSE;
    }

  m_Transfering = 0;
  LEDUpdate();

  GetLocalTime(&TestUSBTime);

  if(hInDevice)
   CloseHandle(hInDevice);
  if(ReadCompleteEvent)
   CloseHandle(ReadCompleteEvent);

  free(outBuffer);
  free(inBuffer);

  return TRUE;

}
```

Función OnBnClickedButtonLoadParam

```
void CResults::OnBnClickedButtonLoadParam()
{

  MessageBox("Beginning the transmission of the parameters.");
  m_cntlDirection.SetWindowTextA("<=====");

  if(SendTrama())
   MessageBox("Parameters file sent.");
```

```
  else
   MessageBox("Sent of Parameters failed.");

}
```

Función OnBnClickedBtnExecute

```
void CResults::OnBnClickedBtnExecute()
{

  ULONG MaxTransferSize;
  FILE * outFile;
  UINT32 SECOND2LAST_FF;
  UINT32 continua, h;
  UINT8 Primera = 0;
  UINT8 Command = START_EMULATION;
  SYSTEMTIME FinishTime;
  MSG msg;

  // Get size of file to transfer
  MaxTransferSize = 0x00000400;

  SECOND2LAST_FF = 0X00000000;

  // Initilize Progress Bar
  m_cntlProgressBar.SetRange32(0,100);
  m_cntlProgressBar.SetBkColor(0x00FFFFFF);
  m_cntlProgressBar.SetBarColor(0x0000C800);
  m_cntlProgressBar.SetPos(0);

  continua = 0x00000000;
  h = 0x00000000;

  if((outFile = fopen(m_strResultsFile, "w")) == NULL)
   MessageBox("Not available file for writting.");
  else{
   GetLocalTime(&IniTime);

  Command = SHOW_RESULTS;
  // Get init value of registers
  m_cntlDirection.SetWindowTextA("=====>");
  ReadTrama(Primera, 0xFFFFFFFF, outFile, Command);
  m_cntlDirection.SetWindowTextA("----");
  Primera = 1;
  UpdateData(TRUE);

   m_clear_button.EnableWindow(false);
   m_show_results.EnableWindow(false);
   m_config_button.EnableWindow(false);
   m_load_button.EnableWindow(false);
   m_start_button.EnableWindow(false);

   Command = START_EMULATION;
   while(h != 0xFFFFFFFF){
```

```
    // Get emulation results
    m_cntlDirection.SetWindowTextA("=====>");
    continua = ReadTrama(Primera, h, outFile, Command);
    m_cntlDirection.SetWindowTextA("----");
    h = continua;
    if (continua != 0xFFFFFFFF)
    SECOND2LAST_FF = continua;
    while( PeekMessage(&msg,NULL,0,0,NULL) != 0) {
     if (GetMessage( &msg, NULL, 0, 0 ) > 0) {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
     }
   }
  }

  m_clear_button.EnableWindow(true);
  m_show_results.EnableWindow(true);
  m_config_button.EnableWindow(true);
  m_load_button.EnableWindow(true);
  m_start_button.EnableWindow(true);


  Command = LAST_RESULT;
  // Get last result
  while(continua != 0x00000000){
   // Get emulation results
   m_cntlDirection.SetWindowTextA("=====>");
   continua = ReadTrama(Primera, SECOND2LAST_FF, outFile,
Command);
   m_cntlDirection.SetWindowTextA("----");
   UpdateData(TRUE);
  }

  GetLocalTime(&FinishTime);
  fprintf(outFile, "\n\nHora de fin de la emulacion:
%02d:%02d\n", FinishTime.wHour,
                FinishTime.wMinute);

  fclose(outFile);
  MessageBox("Emulation finished.");
}
m_cntlDirection.SetWindowTextA("----");

UpdateData(TRUE);

}
```

Función ConfigureUSB

```
BOOLEAN CResults::ConfigureUSB(BOOLEAN bDisplayMsg)
{

  if(SendResetHold()){
   if(SendIntelHexFile()){
```

```
    if(SendCommReset(FALSE)){
     if(bDisplayMsg)
      MessageBox("FX2 Configuration Done");
      return TRUE;
    }
   }
  }

 return FALSE;

}
```

Función SendResetHold

```
BOOLEAN CResults::SendResetHold()
{

  return SendCommReset(TRUE);

}
```

Función SendCommReset

```
BOOLEAN CResults::SendCommReset(BOOLEAN DeviceReset)
{

  VENDOR_REQUEST_IN DeviceRequest;
  HANDLE hOutDevice, WriteCompleteEvent;
  THREAD_CONTROL outThreadControl;
  char DeviceName[MAX_DRIVER_NAME];

  m_strDeviceNumber = _T("0");

  // Setup event object
  WriteCompleteEvent = CreateEvent(0,FALSE,FALSE,NULL);

  // Open device
  sprintf(DeviceName, "EZUSB-" + m_strDeviceNumber);
  if (bOpenDriver (&hOutDevice, DeviceName) != TRUE)
  {
   hOutDevice = NULL;
   CloseHandle(WriteCompleteEvent);
   MessageBox("SendCommReset()->Failed to Open Device");
   return FALSE;
  }

  DeviceRequest.bRequest = 0xA0;
  DeviceRequest.wValue = 0xE600;
  DeviceRequest.wIndex = 0x0;
  DeviceRequest.wLength = 0x01;
  DeviceRequest.bData = (DeviceReset) ? 1 : 0;
  DeviceRequest.direction = 0x0;
```

```
  outThreadControl.hDevice = hOutDevice;

  outThreadControl.Ioctl = IOCTL_Ezusb_VENDOR_REQUEST;
  outThreadControl.InBuffer = (PVOID)&DeviceRequest;
  outThreadControl.InBufferSize = sizeof(VENDOR_REQUEST_IN);
  outThreadControl.OutBuffer = NULL;
  outThreadControl.OutBufferSize = 0;

  outThreadControl.completionEvent = WriteCompleteEvent;
  outThreadControl.status = FALSE;
  outThreadControl.BytesReturned = 0;

  // Start transfer
  AfxBeginThread((AFX_THREADPROC)TransferThread,
                    (LPVOID)&outThreadControl);

  // Wait for complete event
  WaitForSingleObject(WriteCompleteEvent,INFINITE);

  CloseHandle(hOutDevice);
  CloseHandle(WriteCompleteEvent);

  // Check if write failed
  if (!outThreadControl.status)
  {
   MessageBox("SendCommReset()->Error: Write failed");
   return FALSE;
  }

  return TRUE;

}
```

Función SendIntelHexFile

```
BOOLEAN CResults::SendIntelHexFile()
{

  FILE * inFile;
  ANCHOR_DOWNLOAD_CONTROL downloadControl;
  THREAD_CONTROL outThreadControl;
  HANDLE WriteCompleteEvent;
  HANDLE  hOutDevice;
  ULONG MaxTransferSize, CurrentTransferSize;
  ULONG OutPipeNum;
  PUCHAR outBuffer;
  char DeviceName[MAX_DRIVER_NAME];
  char record[50], byte[3], noof_bytes[3], addr[5];
  UINT next_addr, last_addr, start_addr, cntr, buffer_size;

  // Get pipe number (Assumption for ADS demos - Output always
pipe 0)
  OutPipeNum = 0;
```

```
  MaxTransferSize = 0;
  buffer_size = 0;
  next_addr = 0;
  last_addr = 0;
  start_addr = 0;
  cntr = 0;
  noof_bytes[2] = '\0';
  addr[4] = '\0';
  byte[2] = '\0';

  sprintf(DeviceName, "EZUSB-" + m_strDeviceNumber);
  outBuffer = (UCHAR *)malloc(_8K);

  //
  // get the transfer size and allocate the transfer buffers
  //

  if((inFile = fopen(m_strConfigFile, "r")) == NULL)
  {
   free(outBuffer);
   MessageBox("Couldn't open file for reading");
   return FALSE;
  }

  // Setup event object
  WriteCompleteEvent = CreateEvent(0,FALSE,FALSE,NULL);


  // Open device
  if (bOpenDriver (&hOutDevice, DeviceName) != TRUE)
  {
   hOutDevice = NULL;
   CloseHandle(WriteCompleteEvent);
   fclose(inFile);
   free(outBuffer);
   MessageBox("Failed to Open Device");
   return FALSE;
  }

  // Transfer to whole file
  while(!feof(inFile))
  {
    //Get a line
    if(fgets(record, 45, inFile))
    {
     strncpy(noof_bytes, &record[1], 2);
     strncpy(addr, &record[3], 4);
     next_addr = hexstr2uint(addr);
     cntr = hexstr2uint(noof_bytes);
     MaxTransferSize += cntr;   // Amount of data to transfer
     if(!buffer_size)      // Each buffer has contiguous addrs
      start_addr = next_addr;
     while(cntr)           // Fill the output buffer with the
currect record's data
     {
```

```
      strncpy(byte, &record[(hexstr2uint(noof_bytes) - cntr--) *
2 + 9], 2);
      outBuffer[buffer_size++] = hexstr2uint(byte);
     }
     // Check for contiguous addresses
     if(last_addr + hexstr2uint(noof_bytes) != next_addr)
     {                    // Addresses were not contiguous
      if(hOutDevice)      // Check if driver was opened
      {                   // Driver was opened OK
        downloadControl.Offset = start_addr;
       CurrentTransferSize = buffer_size;
       // Make sure data is going to internal memory
       if(start_addr < _8K)
       {
       // initialize data structures for the write thread
       outThreadControl.hDevice = hOutDevice;

       outThreadControl.Ioctl = IOCTL_EZUSB_ANCHOR_DOWNLOAD;
       outThreadControl.InBuffer = (PVOID)&downloadControl;
       outThreadControl.InBufferSize =
sizeof(ANCHOR_DOWNLOAD_CONTROL);
       outThreadControl.OutBuffer = outBuffer;
       outThreadControl.OutBufferSize = CurrentTransferSize;

       outThreadControl.completionEvent = WriteCompleteEvent;
       outThreadControl.status = FALSE;
       outThreadControl.BytesReturned = 0;

       // Start transfer

      AfxBeginThread((AFX_THREADPROC)TransferThread,
(LPVOID)&outThreadControl);

      // Wait for complete event

     WaitForSingleObject(WriteCompleteEvent,INFINITE);

     // Check if write failed
     if (!outThreadControl.status)
     {
      fclose(inFile);
      free(outBuffer);
      CloseHandle(hOutDevice);
      MessageBox("Error: Write failed");
      return FALSE;
     }
    }
    else
     MessageBox("Invalid 8051 address.");
    buffer_size = 0;
    MaxTransferSize = 0;
   }
  }

 }
}
```

```
  fclose(inFile);
  free(outBuffer);
  CloseHandle(hOutDevice);
  CloseHandle(WriteCompleteEvent);

  return TRUE;

}
```

Función TransferThread

```
void TransferThread(PTHREAD_CONTROL threadControl)
{

  // perform the ISO transfer
  threadControl->status = DeviceIoControl (threadControl-
>hDevice,

threadControl->Ioctl,

threadControl->InBuffer,

threadControl->InBufferSize,

threadControl->OutBuffer,

threadControl->OutBufferSize,

&threadControl->BytesReturned,

NULL);

  // if an event exists, set it
  if (threadControl->completionEvent)
   SetEvent(threadControl->completionEvent);

return;

}
```

Función hexstr2uint

```
UINT CResults::hexstr2uint(const char * hexstrptr)
{

  unsigned int charctr, maxchars, byte, result = 0;
  char digit;

  maxchars = (strlen(hexstrptr) > 8) ? 8 : strlen(hexstrptr);
  charctr = 0;

  while(maxchars--)
  {
```

```
   digit = toupper(hexstrptr[charctr]);
   switch (digit)
   {
    case 'F':
    case 'E':
    case 'D':
    case 'C':
    case 'B':
    case 'A':
     byte = digit - 'A' + 10;
    break;
    case '9':
    case '8':
    case '7':
    case '6':
    case '5':
    case '4':
    case '3':
    case '2':
    case '1':
    case '0':
     byte = digit - '0';
    break;
    default:
     return 0;
   }

   if(byte)
  {
   result += byte << (strlen(hexstrptr) - charctr - 1) * 4;
   charctr++;
  }

  return result;

}
```

Función bOpenDriver

```
BOOLEAN CResults::bOpenDriver(HANDLE *phDeviceHandle, PCHAR
devname)
{

  char completeDeviceName[64] = "";
  char pcMsg[64] = "";

  strcat (completeDeviceName, "\\\\.\\");

   strcat (completeDeviceName, devname);

   *phDeviceHandle = CreateFile(completeDeviceName,

GENERIC_WRITE,
```

```
FILE_SHARE_WRITE,
                                                        NULL,

OPEN_EXISTING,
                                                        0,
                                                        NULL);

  if (*phDeviceHandle == INVALID_HANDLE_VALUE) {
   return (FALSE);
  }
  else {
   return (TRUE);
  }

}
```

Función SendTrama

```
BOOLEAN CResults::SendTrama()
{

  FILE * inFile;
  BULK_TRANSFER_CONTROL outBulkControl;
  THREAD_CONTROL outThreadControl;
  HANDLE WriteCompleteEvent;
  HANDLE hOutDevice;
  ULONG MaxTransferSize;
  ULONG outXferSize;
  ULONG OutPipeNum;
  ULONG i;
  ULONG j;
  PUCHAR outBuffer;
  char DeviceName[MAX_DRIVER_NAME];
  BYTE checksum = 0;
  char byte1[3], byte2[3], byte3[3], byte4[3];

  m_cntlDirection.SetWindowTextA("<=====");

  // Block size is 4.5K
  // block_size = (cmd == READ) ? 0x8000: _4K + _1K/2;
  sprintf(DeviceName, "EZUSB-" + m_strDeviceNumber);
  outXferSize = SIZE_OF_HEADER;

  // Open device
  if (bOpenDriver (&hOutDevice, DeviceName) != TRUE)
  {
    hOutDevice = NULL;
    m_charLEDStatus &= ~1;
    MessageBox("SendTrama()->Failed to Open OutDevice handle");
    m_Connected = 0;
    LEDUpdate();
    return FALSE;
   }
```

```
   else{
    m_Transfering = 1;
    LEDUpdate();
   }

  // Setup event object
  WriteCompleteEvent = CreateEvent(0,FALSE,FALSE,NULL);

  // Get pipe number (Default output pipe for ADS Boards)
  OutPipeNum = 0;

  // Get size of file to transfer
  MaxTransferSize = 0;     // Clear the MaxTransferSize

  if((inFile = fopen(m_strParamFile, "rb")) == NULL)
  {
   MessageBox("Couldn't open file Parametros.txt");
   return FALSE;
  }

  while(!feof(inFile))     // Count the bytes in the file
  {
   if (getc(inFile) == '\n')
   {
    MaxTransferSize = MaxTransferSize + 4;
   }
  }
  rewind(inFile);          // Go to start of file

  // Create the output buffer
  outXferSize = MaxTransferSize + SIZE_OF_HEADER + 4 ;     //
Size of data + Max header
  outBuffer = (UCHAR *)malloc(outXferSize);

  //  Init output buffer header
  for(i = 0; i < outXferSize; i++)
   outBuffer[i] = 0;

  // Set Header
  outBuffer[0] = WRITE;               // Byte 0  command
  outBuffer[1] = 0x00;                // Byte 1  command
  outBuffer[2] = outXferSize;         // Byte 2  transfer size
LSB
  outBuffer[3] = 0x00;                // Byte 3  transfer size
MSB
  outBuffer[4] = 0x00;                // Byte 4  reserved
  outBuffer[5] = 0x00;                // Byte 5  reserved
  outBuffer[6] = 0x00;                // Byte 6  reserved
  outBuffer[7] = 0x00;                // Byte 7  reserved
  outBuffer[8] = 0x00;                // Byte 8  reserved
  outBuffer[9] = 0x00;                // Byte 9  reserved
  outBuffer[10] = 0x00;               // Byte 10 reserved
  outBuffer[11] = 0x00;               // Byte 11 reserved

  // Determine starting location of data
  j = SIZE_OF_HEADER;           // Data start offset/Header size
```

```
  MaxTransferSize = 0;

  char str[256];

  while (fgets(str,255,inFile) != NULL)
  {
   str[8] = '\0';
   byte1[0] = str[0];
   byte1[1] = str[1];
   byte1[2] = '\0';
   outBuffer[j+1] = hexstr2uint(byte1);
   byte2[0] = str[2];
   byte2[1] = str[3];
   byte2[2] = '\0';
   outBuffer[j] = hexstr2uint(byte2);
   byte3[0] = str[4];
   byte3[1] = str[5];
   byte3[2] = '\0';
   outBuffer[j+3] = hexstr2uint(byte3);
   byte4[0] = str[6];
   byte4[1] = str[7];
   byte4[2] = '\0';
   outBuffer[j+2] = hexstr2uint(byte4);
   j = j + 4;
  }

  checksum = 0;
  // initialize data structures for the write thread
  outBulkControl.pipeNum = OutPipeNum;
  outThreadControl.hDevice = hOutDevice;
  outThreadControl.Ioctl = IOCTL_EZUSB_BULK_WRITE;
  outThreadControl.InBuffer = (PVOID)&outBulkControl;
  outThreadControl.InBufferSize = sizeof(BULK_TRANSFER_CONTROL);

  outThreadControl.OutBufferSize = outXferSize;
  outThreadControl.OutBuffer = outBuffer;

  outThreadControl.completionEvent = WriteCompleteEvent;
  outThreadControl.status = FALSE;
  outThreadControl.BytesReturned = 0;

  m_Transfering = 1;
  LEDUpdate();

  // Start transfer
  AfxBeginThread((AFX_THREADPROC)TransferThread,
(LPVOID)&outThreadControl);

  // Wait for complete event
  WaitForSingleObject(WriteCompleteEvent,INFINITE);

  m_Transfering = 0;
  LEDUpdate();

  // Check if write failed
```

```
  if (!outThreadControl.status)
     {
   if(hOutDevice)
    CloseHandle(hOutDevice);
   if(WriteCompleteEvent)
    CloseHandle(WriteCompleteEvent);
   MessageBox("SendTrama()->Error: Write failed");
   return FALSE;
  }

  free(outBuffer);
  fclose(inFile);

  if(hOutDevice)
   CloseHandle(hOutDevice);
  if(WriteCompleteEvent)
   CloseHandle(WriteCompleteEvent);

  return TRUE;

}
```

Función ReadTrama

```
UINT32 CResults::ReadTrama(UINT8 Primera, UINT32 Count, FILE
*outFile, UINT8 Command)
{

  BULK_TRANSFER_CONTROL outBulkControl, inBulkControl;
  THREAD_CONTROL outThreadControl, inThreadControl;
  HANDLE WriteCompleteEvent, ReadCompleteEvent;
  HANDLE hOutDevice, hInDevice;
  ULONG inXferSize, outXferSize;
  ULONG OutPipeNum, InPipeNum;
  ULONG i, j;
  PUCHAR outBuffer, inBuffer;
  char DeviceName[MAX_DRIVER_NAME];
  BOOLEAN rcvStatus = TRUE;
  BYTE checksum = 0;
  BYTE ByteRcv[256];
  UINT8 FIN;
  ULONG CuentaPalabras;
  SYSTEMTIME CurrentTime;
  UINT INJ_FF, INJ_CYCLE, Silent, Failure, Latent, RAMLat,
ClkCount1, ClkCount0;
  UINT TOTAL, CurrentHour, CurrentMin, CurrentSec, Percent;
  UINT32 SecondsTime;
  UINT32 CORRECT_FF;
  UINT32 CORRECT_CYCLE;
  UINT32 CYCLE_END;
  UINT32 CLOCK_COUNT_LOW;
  UINT8 UPDATE_SCREEN;
  CString CurrentFF, CurrentCycle, Silents, Failures, Latents,
Total, ETAHour,
```

```
ETAMin, ETASec, ETAPercent;

BYTE Last_Result = 0x00;

CString Registers;
UINT32 ReferHour, ReferMinute, ReferSeconds, EndTime;

CORRECT_FF = 0;
CORRECT_CYCLE = 0;
CYCLE_END = 0;
CLOCK_COUNT_LOW = 0;
UPDATE_SCREEN = 0;
INJ_FF = 0;
INJ_CYCLE = 0;
Silent = 0;
Failure = 0;
Latent = 0;
RAMLat = 0;
ClkCount1 = 0;
ClkCount0 = 0;
TOTAL = 0;
CurrentHour = 0;
CurrentMin = 0;
CurrentSec = 0;
Percent = 0;
BYTE order = 0x00;

CuentaPalabras = 0;
for (i = 0; i < 256; i++)
 ByteRcv[i] = 0;

m_cntlDirection.SetWindowTextA("=====>");

m_strDeviceNumber = _T("0"); // Set to 0 value for debug

sprintf(DeviceName, "EZUSB-" + m_strDeviceNumber);
inXferSize = RETURN_BYTES;
outXferSize = SIZE_OF_HEADER;

// Open device
if (bOpenDriver (&hOutDevice, DeviceName) != TRUE)
{
 hOutDevice = NULL;
 m_charLEDStatus &= ~1;
 MessageBox("ReadTrama()->Failed to Open OutDevice handle");
 m_Connected = 0;
 LEDUpdate();
 return FALSE;
}
else{
 m_Connected = 1;
 LEDUpdate();
}
if (bOpenDriver(&hInDevice, DeviceName) != TRUE)
{
 hInDevice = NULL;
```

```
   m_charLEDStatus &= ~1;
   if(hOutDevice)
    CloseHandle(hOutDevice);
   MessageBox("ReadTrama()->Failed to Open InDevice handle");
   m_Connected = 0;
   LEDUpdate();
   return FALSE;

  }
  else{
   m_Connected = 1;
   LEDUpdate();
  }
  // Setup event object
  WriteCompleteEvent = CreateEvent(0,FALSE,FALSE,NULL);
  ReadCompleteEvent = CreateEvent(0,FALSE,FALSE,NULL);

  // Get pipe number (Default output pipe for ADS Boards)
  OutPipeNum = 0;
  InPipeNum = 2;

  outBuffer = (UCHAR *)malloc(outXferSize);
  inBuffer = (UCHAR *)malloc(inXferSize);

  for(i = 0; i < outXferSize; i++)
   outBuffer[i] = 0;

  for(i = 0; i < inXferSize; i++)
   inBuffer[i] = 0;

  switch (Command)
  {
   case START_EMULATION:
    order = READ;
   break;
   case CLEAR_REGISTERS:
    order = CLEAR;
   break;
   case SHOW_RESULTS:
    order = SHOW;
   break;
   case LAST_RESULT:
    order = SHOW;
   break;
   default:
    MessageBox("Wrong instruction to send.");
   }

  // Load instruction for receive transmision
  // Set Header
  outBuffer[0] = order;              // Byte 0  command
  outBuffer[1] = 0x00;               // Byte 1  command
  outBuffer[2] = 0x00;               // Byte 2  transfer size
LSB
  outBuffer[3] = 0x01;               // Byte 3  transfer size
MSB
```

```
  outBuffer[4] = 0x00;                // Byte 4  reserved
  outBuffer[5] = 0x00;                // Byte 5  reserved
  outBuffer[6] = Count;               // Byte 6  reserved
  outBuffer[7] = 0x00;                // Byte 7  reserved
  outBuffer[8] = 0x00;                // Byte 8  reserved
  outBuffer[9] = 0x00;                // Byte 9  reserved
  outBuffer[10] = 0x00;               // Byte 10 reserved
  outBuffer[11] = 0x00;               // Byte 11 reserved

  checksum = 0;

  // initialize data structures for the write thread
  outBulkControl.pipeNum = OutPipeNum;
  outThreadControl.hDevice = hOutDevice;
  outThreadControl.Ioctl = IOCTL_EZUSB_BULK_WRITE;
  outThreadControl.InBuffer = (PVOID)&outBulkControl;
  outThreadControl.InBufferSize = sizeof(BULK_TRANSFER_CONTROL);

  outThreadControl.OutBuffer = outBuffer;
  outThreadControl.OutBufferSize = outXferSize;

  outThreadControl.completionEvent = WriteCompleteEvent;
  outThreadControl.status = FALSE;
  outThreadControl.BytesReturned = 0;

  m_Transfering = 1;
  LEDUpdate();

  // Start transfer
  AfxBeginThread((AFX_THREADPROC)TransferThread,
(LPVOID)&outThreadControl);

  // Wait for complete event
  WaitForSingleObject(WriteCompleteEvent,INFINITE);

  // Check if write failed
  if (!outThreadControl.status)
  {
    if(hOutDevice)
     CloseHandle(hOutDevice);
    if(WriteCompleteEvent)
     CloseHandle(WriteCompleteEvent);
    MessageBox("SendTrama()->Error: Write failed");
    return FALSE;
   }

  m_Transfering = 0;
  LEDUpdate();

  // initialize data structures for the read thread
  inBulkControl.pipeNum = InPipeNum;
  inThreadControl.hDevice = hInDevice;
  inThreadControl.Ioctl = IOCTL_EZUSB_BULK_READ;
  inThreadControl.InBuffer = (PVOID)&inBulkControl;
  inThreadControl.InBufferSize = sizeof(BULK_TRANSFER_CONTROL);
```

```
  inThreadControl.OutBuffer = &inBuffer[0];
  inThreadControl.OutBufferSize = inXferSize;

  inThreadControl.completionEvent = ReadCompleteEvent;
  inThreadControl.status = FALSE;
  inThreadControl.BytesReturned = 0;

  m_Transfering = 1;
  LEDUpdate();

  // Start transfer
  AfxBeginThread((AFX_THREADPROC)TransferThread,
      (LPVOID)&inThreadControl);

  // Wait for complete event
  WaitForSingleObject(ReadCompleteEvent, INFINITE);

  if (!inThreadControl.status)  // Check if read failed
   {
   DWORD error =  GetLastError();

   if(hInDevice)
    CloseHandle(hInDevice);
   if(ReadCompleteEvent)
    CloseHandle(ReadCompleteEvent);
    MessageBox("ReadTrama()->Error: Read failed");
    return FALSE;
   }

  m_Transfering = 0;
  LEDUpdate();

  // Get data
  for(i = 0; i <= 128; i++)
   ByteRcv[i+1] = inBuffer[i];

  FIN = ByteRcv[1];

  INJ_FF = 16777216 * ByteRcv[74] + 65536 * ByteRcv[73] + 256 *
ByteRcv[76] + ByteRcv[75];
  INJ_CYCLE = 16777216 * ByteRcv[78] + 65536 * ByteRcv[77] + 256
* ByteRcv[80] + ByteRcv[79];
  CYCLE_END = 16777216 * ByteRcv[30] + 65536 * ByteRcv[29] + 256
* ByteRcv[32] + ByteRcv[31];
  CLOCK_COUNT_LOW = 16777216 * ByteRcv[102] + 65536 *
ByteRcv[101] + 256 * ByteRcv[104] + ByteRcv[103];
  IniFF = 16777216 * ByteRcv[18] + 65536 * ByteRcv[17] + 256 *
ByteRcv[20] + ByteRcv[19];
  EndFF = 16777216 * ByteRcv[22] + 65536 * ByteRcv[21] + 256 *
ByteRcv[24] + ByteRcv[23];

  switch (Command)
  {
   case START_EMULATION:
    CORRECT_FF = INJ_FF - 1;
    CORRECT_CYCLE = INJ_CYCLE;
```

```
   break;
   case CLEAR_REGISTERS:
    CORRECT_FF = INJ_FF;
    CORRECT_CYCLE = INJ_CYCLE;
   break;
   case SHOW_RESULTS:
    if (CLOCK_COUNT_LOW == 0x00000000){
     CORRECT_FF = INJ_FF;
     CORRECT_CYCLE = INJ_CYCLE;
    }
    else{
     CORRECT_FF = EndFF;
     CORRECT_CYCLE = CYCLE_END;
    }
   break;
   case LAST_RESULT:
    CORRECT_FF = EndFF;
    CORRECT_CYCLE = CYCLE_END;
   break;
   default:
    CORRECT_FF = INJ_FF;
   break;
  }

  switch (Command)
  {
   case START_EMULATION:
   if (INJ_FF != Count)
    UPDATE_SCREEN = 1;
   else
    UPDATE_SCREEN = 0;
  break;
  case CLEAR_REGISTERS:
   UPDATE_SCREEN = 1;
  break;
  case SHOW_RESULTS:
   UPDATE_SCREEN = 1;
  break;
  case LAST_RESULT:
   if (INJ_FF == 0x00000000)
    UPDATE_SCREEN = 1;
   else
    UPDATE_SCREEN = 0;
   break;
   default:
   break;
  }

   // Escritura en el archivo de almacenaje de datos
   if (Primera == 0){
    GetLocalTime(&CurrentTime);
    fprintf(outFile, "\nHora de comienzo de la emulacion:
%02d:%02d\n", CurrentTime.wHour,
                CurrentTime.wMinute);

    fprintf(outFile, "Parametros de la emulacion: \n");
```

```
    fprintf(outFile, "\n");
    for(j = 0; j < 27; j++)
     fprintf(outFile, " ");
    for(j = 0; j < 7; j++)
     fprintf(outFile, "---------- ");
    fprintf(outFile, "\n");
    for(j = 0; j < 26; j++)
     fprintf(outFile, " ");
    fprintf(outFile, "| RCONTROL |  RSTATUS |  FFBEGIN |  FFEND
| CYCLEBEG | CYCLEEND | TBCYCLES |\n");
     for(j = 0; j < 27; j++)
      fprintf(outFile, " ");
     for(j = 0; j < 7; j++)
      fprintf(outFile, "---------- ");
     fprintf(outFile, "\n");
     for(j = 0; j < 26; j++)
      fprintf(outFile, " ");
      i = 12;
     while(i < 40)
    {
      fprintf(outFile, "| ");
      fprintf(outFile, "%02X", ByteRcv[i-2]);
      fprintf(outFile, "%02X", ByteRcv[i-3]);
      fprintf(outFile, "%02X", ByteRcv[i]);
      fprintf(outFile, "%02X", ByteRcv[i-1]);
      fprintf(outFile, " ");
      i = i + 4;
     }
     fprintf(outFile, "|");

     fprintf(outFile, "\n");
     for(j = 0; j < 27; j++)
      fprintf(outFile, " ");
     for(j = 0; j < 7; j++)
      fprintf(outFile, "---------- ");
     fprintf(outFile, "\n");
     for(j = 0; j < 40; j++)
      fprintf(outFile, " ");
     for(j = 0; j < 4; j++)
      fprintf(outFile, "------------ ");
     fprintf(outFile, "\n");
     for(j = 0; j < 39; j++)
      fprintf(outFile, " ");
     fprintf(outFile, "|    RATE    |   LENGTH   |   OFFSET   |
ITERATIONS |\n");
     for(j = 0; j < 40; j++)
      fprintf(outFile, " ");
     for(j = 0; j < 4; j++)
      fprintf(outFile, "------------ ");
     fprintf(outFile, "\n");
     for(j = 0; j < 39; j++)
      fprintf(outFile, " ");
      i = 52;
     while(i < 64)
    {
     fprintf(outFile, "|  ");
```

```
     fprintf(outFile, "%02X", ByteRcv[i-2]);
     fprintf(outFile, "%02X", ByteRcv[i-3]);
     fprintf(outFile, "%02X", ByteRcv[i]);
     fprintf(outFile, "%02X", ByteRcv[i-1]);
     fprintf(outFile, "  ");
     i = i + 4;
    }
    fprintf(outFile, "|  ");
    fprintf(outFile, "%02X", ByteRcv[66]);
    fprintf(outFile, "%02X", ByteRcv[65]);
    fprintf(outFile, "%02X", ByteRcv[68]);
    fprintf(outFile, "%02X", ByteRcv[67]);
    fprintf(outFile, "  |\n");
    for(j = 0; j < 40; j++)
     fprintf(outFile, " ");
    for(j = 0; j < 4; j++)
     fprintf(outFile, "------------ ");
    fprintf(outFile, "\n");
    for(j = 0; j < 14; j++)
     fprintf(outFile, " ");
    for(j = 0; j < 8; j++)
     fprintf(outFile, "------------ ");
    fprintf(outFile, "\n");
    for(j = 0; j < 13; j++)
     fprintf(outFile, " ");
    fprintf(outFile, "|   INJ_FF   | INJ_CYCLE  |   Silent   |
Failure   |   Latent   |   RAMLat   | ClkCount1  | ClkCount0
|");
    fprintf(outFile, "\n");
    for(j = 0; j < 14; j++)
     fprintf(outFile, " ");
    for(j = 0; j < 8; j++)
     fprintf(outFile, "------------ ");
    fprintf(outFile, "\n");
    for(j = 0; j < 13; j++)
     fprintf(outFile, " ");
    i = 76;
    while(i < 108)
    {
      fprintf(outFile, "|  ");
      fprintf(outFile, "%02X", ByteRcv[i-2]);
      fprintf(outFile, "%02X", ByteRcv[i-3]);
      fprintf(outFile, "%02X", ByteRcv[i]);
      fprintf(outFile, "%02X", ByteRcv[i-1]);
      fprintf(outFile, "  ");
      i = i + 4;
     }
     fprintf(outFile, "|\n");
     for(j = 0; j < 14; j++)
      fprintf(outFile, " ");
     for(j = 0; j < 8; j++)
      fprintf(outFile, "------------ ");
     fprintf(outFile, "\n");
     fprintf(outFile, "\n\nResultados de la emulacion: \n" );
     fprintf(outFile, "\n ");
     for(j = 0; j < 10; j++)
```

```
      fprintf(outFile, "------------ ");
      fprintf(outFile, "\n|  RCONTROL  |   RSTATUS  |   INJ_FF
| INJ_CYCLE  |   Silent   |  Failure   |   Latent   |   RAMLat
| ClkCount1  | ClkCount0  |");
      fprintf(outFile, "\n ");
      for(j = 0; j < 10; j++)
       fprintf(outFile, "------------ ");
     }

     if (UPDATE_SCREEN == 1){
     // Select Bytes with registers information

     fprintf(outFile, "\n");
     i = 12;
     while(i < 20)
    {
      fprintf(outFile, "|  ");
      fprintf(outFile, "%02X", ByteRcv[i-2]);
      fprintf(outFile, "%02X", ByteRcv[i-3]);
      fprintf(outFile, "%02X", ByteRcv[i]);
      fprintf(outFile, "%02X", ByteRcv[i-1]);
      fprintf(outFile, "  ");
      i = i + 4;
     }

     fprintf(outFile, "|  %08X  ", CORRECT_FF);
     fprintf(outFile, "|  %08X  ", CORRECT_CYCLE);
     i = 84;
     while(i < 108)
     {
      fprintf(outFile, "|  ");
      fprintf(outFile, "%02X", ByteRcv[i-2]);
      fprintf(outFile, "%02X", ByteRcv[i-3]);
      fprintf(outFile, "%02X", ByteRcv[i]);
      fprintf(outFile, "%02X", ByteRcv[i-1]);
      fprintf(outFile, "  ");

      switch (i){
       case 84:
        Silent = 16777216 * ByteRcv[i-2] + 65536 * ByteRcv[i-3]
+ 256 * ByteRcv[i] + ByteRcv[i-1];
       break;
       case 88:
        Failure = 16777216 * ByteRcv[i-2] + 65536 * ByteRcv[i-3]
+ 256 * ByteRcv[i] + ByteRcv[i-1];
       break;
       case 92:
        Latent = 16777216 * ByteRcv[i-2] + 65536 * ByteRcv[i-3]
+ 256 * ByteRcv[i] + ByteRcv[i-1];
       break;
       case 96:
        RAMLat = 16777216 * ByteRcv[i-2] + 65536 * ByteRcv[i-3]
+ 256 * ByteRcv[i] + ByteRcv[i-1];
       break;
       case 100:
```

```
        ClkCount1 = 16777216 * ByteRcv[i-2] + 65536 * ByteRcv[i-
3] + 256 * ByteRcv[i] + ByteRcv[i-1];
       break;
       case 104:
        ClkCount0 = 16777216 * ByteRcv[i-2] + 65536 * ByteRcv[i-
3] + 256 * ByteRcv[i] + ByteRcv[i-1];
       break;
       default:
       break;
      }
      i = i + 4;
     }

     fprintf(outFile, "|");

     TOTAL = Silent + Failure + Latent;

     ReferHour = IniTime.wHour;
     ReferMinute = IniTime.wMinute;
     ReferSeconds = IniTime.wSecond;

     GetLocalTime(&CurrentTime);

     CurrentHour = CurrentTime.wHour;
     CurrentMin = CurrentTime.wMinute;
     CurrentSec = CurrentTime.wSecond;

     SecondsTime = CurrentHour * 3600 + CurrentMin * 60 +
CurrentSec
                        - (ReferHour * 3600 + ReferMinute * 60 +
ReferSeconds);

     if (CORRECT_FF == IniFF)
      EndTime = 359999;
     else{
      EndTime = SecondsTime * (EndFF - IniFF) / (CORRECT_FF -
IniFF);
      EndTime = EndTime - SecondsTime;
     }

     CurrentHour = EndTime / 3600;
     CurrentMin = (EndTime - CurrentHour * 3600) / 60;
     CurrentSec = (EndTime - CurrentHour * 3600 - CurrentMin *
60);

     if ((EndFF - IniFF) == 0)
      Percent = 0;
     else
      Percent = 100 * (CORRECT_FF - IniFF)/ (EndFF - IniFF);

     m_cntlProgressBar.SetPos(Percent);

     CurrentFF.Format("%08X", CORRECT_FF);
     CurrentCycle.Format("%08X", CORRECT_CYCLE);
     Silents.Format("%08X", Silent);
     Failures.Format("%08X", Failure);
```

```
     Latents.Format("%08X", Latent);
     Total.Format("%08X", TOTAL);
     ETAHour.Format("%d", CurrentHour);
     ETAMin.Format("%d", CurrentMin);
     ETASec.Format("%d", CurrentSec);
     ETAPercent.Format("%d", Percent);

     // Registers update
     m_cntlCurrentFF.SetWindowTextA(CurrentFF);
     m_cntlCurrentCycle.SetWindowTextA(CurrentCycle);
     m_cntlSilents.SetWindowTextA(Silents);
     m_cntlFailures.SetWindowTextA(Failures);
     m_cntlLatents.SetWindowTextA(Latents);
     m_cntlTotal.SetWindowTextA(Total);
     m_cntlETAHour.SetWindowTextA(ETAHour);
     m_cntlETAMin.SetWindowTextA(ETAMin);
     m_cntlETASec.SetWindowTextA(ETASec);
     m_cntlETAPercent.SetWindowTextA(ETAPercent);

     Registers.Format(" %08X   %08X     %08X    %08X     %08X
%08X    %08X     %08X",
             CORRECT_FF, CORRECT_CYCLE, Silent, Failure, Latent,
RAMLat, ClkCount1, ClkCount0);

     m_listResults.AddString(Registers);
     m_listResults.SetCurSel(m_listResults.GetCount()-1);

     CResults::RedrawWindow();

    }

    if (Command == START_EMULATION){
     if (INJ_FF == EndFF)
      INJ_FF = 0xFFFFFFFF;
    }

    if(hInDevice)
     CloseHandle(hInDevice);
    if(ReadCompleteEvent)
     CloseHandle(ReadCompleteEvent);

    free(outBuffer);
    free(inBuffer);

    return INJ_FF;

}
```

Función LEDUpdate

```
void CResults::LEDUpdate()
{

  if (m_Connected == 1){
```

```
   m_ConnectedLedOff.ShowWindow(SW_HIDE);
   m_ConnectedLedOn.ShowWindow(SW_SHOW);
  }
  else{
   m_ConnectedLedOff.ShowWindow(SW_SHOW);
   m_ConnectedLedOn.ShowWindow(SW_HIDE);
  }

  if (m_Transfering == 1){
   m_TransferingLedOff.ShowWindow(SW_HIDE);
   m_TransferingLedOn.ShowWindow(SW_SHOW);
  }
  else{
   m_TransferingLedOff.ShowWindow(SW_HIDE);
   m_TransferingLedOn.ShowWindow(SW_SHOW);
  }

  m_ConnectedLedOff.UpdateData();
  m_ConnectedLedOn.UpdateData();
  m_TransferingLedOff.UpdateData();
  m_TransferingLedOn.UpdateData();

}
```

Función OnBnClickedBtnClear

```
void CResults::OnBnClickedBtnClear()
{

  UINT8 Command = CLEAR_REGISTERS;
  FILE * ClearFile;
  UINT32 Show = 0xFFFFFFFF;

  if((ClearFile =
fopen("D:/Pedro/Proyectos_USB_en_C/USB2Util_v5_8/Debug/data/Clea
rFile.deb", "w")) == NULL)
  MessageBox("Not available file for writing.");

  ReadTrama(NULL, Show, ClearFile, Command);
  UpdateData(TRUE);

   fclose(ClearFile);

}
```

Función OnBnClickedBtnShow

```
void CResults::OnBnClickedBtnShow()
{

  UINT8 Command = SHOW_RESULTS;
  FILE * ShowFile;
  UINT32 Show = 0xFFFFFFFF;
```

```
  if((ShowFile =
fopen("D:/Pedro/Proyectos_USB_en_C/USB2Util_v5_8/Debug/data/Show
File.deb", "w")) == NULL)
   MessageBox("Not available file for writing.");

  ReadTrama(NULL, Show, ShowFile, Command);
  UpdateData(TRUE);

  fclose(ShowFile);

}
```

Función ReadConfig

```
BOOLEAN CResults::ReadConfig(void)
{

  FILE *ConfigFile;
  int FileColumn = 0;
  int FileRow = 0;
  char FileContent[33][100];          // File dimension
  int ColumnSize = 0;
  int RowSize = 0;
  BOOLEAN FIN = FALSE;
  int i = 0;
  int j = 0;
  int k = 0;
  int l = 0;
  int str_int = 0;
  const char str_temp_1[2] = ".";
  const char str_temp_2[3]= "\\0";
  const char str_temp_3[2] = "/";
  char char_temp = 0;
  UCHAR * str_var_1;
  UCHAR * str_var_2;
  UCHAR * str_var_3;
  int int_var_1 = 0;
  int int_var_2 = 0;
  int int_var_3 = 0;

  for (i = 0; i < 33; i++){
   for (j = 0; j < 100; j++)
    FileContent[i][j] = ' ';
  }

  i = 0;
  l = 0;
  k = 0;

  if((ConfigFile = fopen("Configuration.cfg", "r")) == NULL)
   MessageBox("Couldn't open file for reading");
  else{
   rewind(ConfigFile);
```

```
// Safe File content
while(!feof(ConfigFile)){
 FileContent[FileRow][FileColumn] = fgetc(ConfigFile);
 if(FileContent[FileRow][FileColumn] == '\n'){
  FileRow++;
  FileColumn = 0;
 }
 else
  switch (FileRow){
   case Config_Row:
    if (i == 0){
     if (FileContent[FileRow][FileColumn] == str_temp_1[0]){
      int_var_1 = int_var_1 + 4;
      i = 1;
     }
    else
     int_var_1++;
    }
   break;
   case Param_Row:
    if (l == 0){
     if (FileContent[FileRow][FileColumn] == str_temp_1[0]){
      int_var_2 = int_var_2 + 4;
      l = 1;
     }
     else
      int_var_2++;
    }
   break;
   case Results_Row:
    if (k == 0){
     if (FileContent[FileRow][FileColumn] == str_temp_1[0]){
      int_var_3 = int_var_3 + 4;
       k = 1;
     }
     else
      int_var_3++;
    }
   break;
   default:
   break;
  }
  FileColumn++;
 }
 rewind(ConfigFile);
 fclose(ConfigFile);
}
RowSize = FileRow;
ColumnSize = FileColumn;

int_var_1 = int_var_1 - 8;
int_var_2 = int_var_2 - 8;
int_var_3 = int_var_3 - 8;

str_var_1 = (UCHAR *)calloc(int_var_1, sizeof(UCHAR *));
str_var_2 = (UCHAR *)calloc(int_var_2, sizeof(UCHAR *));
```

```
str_var_3 = (UCHAR *)calloc(int_var_3, sizeof(UCHAR *));

for (j = 0; j < int_var_1; j++)
 str_var_1[i] = ' ';

for (j = 0; j < int_var_2; j++)
 str_var_2[i] = ' ';

for (j = 0; j < int_var_3; j++)
 str_var_3[i] = ' ';

j = 10;
k = 0;
while(FileContent[4][j] != str_temp_1[0])
{
  if (FileContent[4][j] == str_temp_2[0]){
   str_var_1[j+k-10] = str_temp_3[0];
   str_var_1[j+k-9] = str_temp_3[0];
   k++;
 }
 else{
  str_var_1[j+k-10] = FileContent[4][j];
 }
 j++;
 }
 for (i = j; i < j+4; i++){
  str_var_1[i+k-10] = FileContent[4][i];
 }
 j = 10;
 k = 0;
 while(FileContent[5][j] != str_temp_1[0])
 {
  if (FileContent[5][j] == str_temp_2[0]){
   str_var_2[j+k-10] = str_temp_3[0];
   str_var_2[j+k-9] = str_temp_3[0];
   k++;
  }
  else{
   str_var_2[j+k-10] = FileContent[5][j];
  }
  j++;
 }
 for (i = j; i < j+4; i++){
  str_var_2[i+k-10] = FileContent[5][i];
 }
 j = 10;
 k = 0;
 while(FileContent[6][j] != str_temp_1[0])
 {
  if (FileContent[6][j] == str_temp_2[0]){
   str_var_3[j+k-10] = str_temp_3[0];
   str_var_3[j+k-9] = str_temp_3[0];
   k++;
  }
  else{
   str_var_3[j+k-10] = FileContent[6][j];
```

```
      }
      j++;
     }
     for (i = j; i < j+4; i++){
      str_var_3[i+k-10] = FileContent[6][i];
     }

     m_strConfigFile = str_var_1;
     m_strParamFile = str_var_2;
     m_strResultsFile = str_var_3;

     return TRUE;

}
```

BIBLIOGRAFÍA

BIBLIOGRAFÍA

TESIS DOCTORALES:

"Técnicas de inyección de fallos basadas en FPGAs para la evaluación de la tolerancia a fallos de tipo SEU en circuitos digitales". Marta Portela García. Año 2007.

ARTÍCULOS:

"Techniques for Fast Transient Fault Grading Based on Autonomous Emulation". Celia López Ongil, Mario García Valderas, Marta Portela García, Luis Entrena Arrontes. Design, Automation and Test in Europe, 2005. Proceedings. 7-11 de Marzo de 2005. Páginas 308 - 309 Vol. 1

"Autonomous Fault Emulation: A New FPGA-Based Acceleration System for Hardness Evaluation". Celia López Ongil, Mario García Valderas, Marta Portela García, Luis Entrena Arrontes. Nuclear Science, IEEE Transactions on. Febrero 2007. Páginas 252 – 261 Vol. 54.

"Study of SEU Effects in a Turbo Decoder Bit Error Rate". M. Portela Garcia, M. Garcia Valderas, C. López Ongil, L. Entrena, B. Lestriez, L. Berrojo. Test Workshop, 2009. LATW '09. 10th Latin American. Año 2009. Páginas 1 – 5.

"Closing in on the perfect code". Erico Guizzo. Spectrum, IEEE. Marzo 2004. Páginas 36 – 42 Vol. 41.

"Analysis of Turbo Decoder Robustness. Against SEU Effects". Marta Portela García, Celia López Ongil, Mario García Valderas, Luis Entrena, Bruno Lestriez, and Luis Berrojo. Nuclear Science, IEEE Transactions on. Año 2009. Páginas 2184 – 2188. Vol. 56.

MANUALES:

"Xilinx Virtex-4 LX Evaluation Kit UserGuide". Avnet, Inc. Rev 2.0 03/07/2007

"Embedded System Tools Reference Manual. Embedded Development Kit, EDK 8.2i". Xilinx, Inc. UG111 (v6.0) June 23, 2006.

"MicroBlaze Processor Reference Guide. Embedded Development Kit EDK 8.2i". Xilinx, Inc. UG081 (v6.3) August 29, 2006.

"ADS USB 2.0 Utility User Manual". Avnet, Inc. June 8, 2005.

"EDK OS and Libraries Reference Guide". Xilinx Inc. UG114 (v3.0) June 22, 2004.

"How to Create and Program Interrupt-Based Systems", Jason Agron. 01/08/2008.

HOJAS DE CARACTERÍSTICAS:

CY7C68013 datasheet. Cypress Semiconductor, Inc. Rev. E. February 8, 2005

OPB UART Lite (v1.00b). Xilinx, Inc. DS422 (v1.00b) July 24, 2006.

OPB General Purpose Input/Output (GPIO). Xilinx, Inc. DS466 (v3.01b) December 1, 2005

OPB External Memory Controller (OPB EMC). Xilinx, Inc. DS421 (2.00a) January 16, 2006.

OPB Universal Serial Bus 2.0 Device. Xilinx, Inc. DS591 (v1.00a) May 10, 2007.

LogiCORE IP On-Chip Peripheral Bus V2.0 with OPB Arbiter. Xilinx, Inc. DS402 (v1.00d) April 19, 2010.

NOTAS TÉCNICAS:

Título:	**Autor/es:**
Construcción de emuladores de fallos	Mario García Valderas
Quick StartV4	AVNET Design Resource Center
Using and Creating Interrupt-Based Systems	Paul Glover

DIRECCIONES DE PÁGINAS DE INTERNET:

Página oficial de Xilinx:

http://www.xilinx.com

AVNET Design Resource Center:

http://www.em.avnet.com

Asociación IEEE:

http://www.ieee.org